

TCP, UDP, and Sockets:
rigorous and experimentally-validated behavioural specification

Volume 2: The Specification

Steve Bishop*
Matthew Fairbairn*
Michael Norrish†
Peter Sewell*
Michael Smith*
Keith Wansbrough*

*University of Cambridge Computer Laboratory

†NICTA, Canberra

March 18, 2005

Brief Contents

| | |
|---|------------|
| Brief Contents | i |
| How to read this document | iv |
| Full Contents | v |
| 1 Utility functions | 2 |
| 2 Error codes | 7 |
| 3 Signal names | 10 |
| 4 Base types | 13 |
| 5 Network datagram types | 25 |
| 6 System call types | 33 |
| 7 Host LTS labels and rule categories | 38 |
| 8 Rule names | 42 |
| 9 Timers | 45 |
| 10 Host types | 53 |
| 11 Host behavioural parameters | 66 |
| 12 Auxiliary functions | 79 |
| 13 Relational monad | 103 |
| 14 Auxiliary functions for TCP segment creation and drop | 106 |
| 15 Host LTS: Socket Calls | 124 |
| 15.1 accept() (TCP only) | 124 |
| 15.2 bind() (TCP and UDP) | 130 |
| 15.3 close() (TCP and UDP) | 136 |
| 15.4 connect() (TCP and UDP) | 145 |
| 15.5 disconnect() (TCP and UDP) | 161 |
| 15.6 dup() (TCP and UDP) | 166 |
| 15.7 dupfd() (TCP and UDP) | 168 |
| 15.8 getfileflags() (TCP and UDP) | 170 |
| 15.9 getifaddrs() (TCP and UDP) | 172 |
| 15.10 getpeername() (TCP and UDP) | 173 |
| 15.11 getsockbopt() (TCP and UDP) | 176 |
| 15.12 getsockopterr() (TCP and UDP) | 179 |
| 15.13 getsocklistening() (TCP and UDP) | 181 |
| 15.14 getsockname() (TCP and UDP) | 183 |
| 15.15 getsocknopt() (TCP and UDP) | 187 |
| 15.16 getsocktopt() (TCP and UDP) | 189 |
| 15.17 listen() (TCP only) | 191 |
| 15.18 pselect() (TCP and UDP) | 198 |
| 15.19 recv() (TCP only) | 206 |

| | | |
|-----------|--|------------|
| 15.20 | recv() (UDP only) | 218 |
| 15.21 | send() (TCP only) | 229 |
| 15.22 | send() (UDP only) | 239 |
| 15.23 | setfileflags() (TCP and UDP) | 253 |
| 15.24 | setsockbopt() (TCP and UDP) | 255 |
| 15.25 | setsocknopt() (TCP and UDP) | 257 |
| 15.26 | setsockopt() (TCP and UDP) | 260 |
| 15.27 | shutdown() (TCP and UDP) | 263 |
| 15.28 | socketmark() (TCP only) | 267 |
| 15.29 | socket() (TCP and UDP) | 271 |
| 16 | Host LTS: TCP Input Processing | 278 |
| - | <i>deliver_in_1</i> | 279 |
| - | <i>deliver_in_1b</i> | 283 |
| - | <i>deliver_in_2</i> | 285 |
| - | <i>deliver_in_2a</i> | 290 |
| - | <i>deliver_in_3</i> | 291 |
| - | <i>deliver_in_3a</i> | 309 |
| - | <i>deliver_in_3b</i> | 310 |
| - | <i>deliver_in_3c</i> | 311 |
| - | <i>deliver_in_4</i> | 312 |
| - | <i>deliver_in_5</i> | 313 |
| - | <i>deliver_in_6</i> | 313 |
| - | <i>deliver_in_7</i> | 314 |
| - | <i>deliver_in_7a</i> | 315 |
| - | <i>deliver_in_7b</i> | 316 |
| - | <i>deliver_in_7c</i> | 317 |
| - | <i>deliver_in_7d</i> | 318 |
| - | <i>deliver_in_8</i> | 319 |
| - | <i>deliver_in_9</i> | 320 |
| 17 | Host LTS: TCP Output | 322 |
| - | <i>deliver_out_1</i> | 323 |
| 18 | Host LTS: TCP Timers | 325 |
| - | <i>timer_tt_rexmtsyn_1</i> | 325 |
| - | <i>timer_tt_rexmt_1</i> | 327 |
| - | <i>timer_tt_persist_1</i> | 329 |
| - | <i>timer_tt_keep_1</i> | 329 |
| - | <i>timer_tt_2msl_1</i> | 330 |
| - | <i>timer_tt_delack_1</i> | 331 |
| - | <i>timer_tt_conn_est_1</i> | 331 |
| - | <i>timer_tt_fin_wait_2_1</i> | 331 |
| 19 | Host LTS: UDP Input Processing | 333 |
| - | <i>deliver_in_udp_1</i> | 333 |
| - | <i>deliver_in_udp_2</i> | 333 |
| - | <i>deliver_in_udp_3</i> | 334 |
| 20 | Host LTS: ICMP Input Processing | 335 |
| - | <i>deliver_in_icmp_1</i> | 335 |
| - | <i>deliver_in_icmp_2</i> | 336 |
| - | <i>deliver_in_icmp_3</i> | 337 |
| - | <i>deliver_in_icmp_4</i> | 338 |
| - | <i>deliver_in_icmp_5</i> | 339 |
| - | <i>deliver_in_icmp_6</i> | 339 |
| - | <i>deliver_in_icmp_7</i> | 340 |
| 21 | Host LTS: Network Input and Output | 341 |
| - | <i>deliver_in_99</i> | 341 |
| - | <i>deliver_in_99a</i> | 341 |
| - | <i>deliver_out_99</i> | 341 |
| - | <i>deliver_loop_99</i> | 342 |
| 22 | Host LTS: BSD Trace Records and Interface State Changes | 343 |

23 Host LTS: Time Passage

345

24 Initial state

351

Index

354

How to read this document

This document is a rigorous specification of the behaviour of TCP, UDP, and the Sockets interface, experimentally validated against the behaviour of several implementations. It is written in the higher order logic of the HOL system.

For a full discussion of the specification we refer the reader to the companion *Volume 1: Overview* and especially to the section there titled “The Specification — Introduction”, which gives a brief introduction to the HOL language and to the structure of the model.

The specification is organised as a reference (in approximately the logical order in which it is presented to the HOL system), not as a tutorial. To read it one should first look at the key types used (base types, network datagram types, and host types) and then browse the Host LTS Socket Call rules and TCP and UDP input and output processing rules.

Full Contents

| | |
|-----------------------------|----------|
| Brief Contents | i |
| How to read this document | iv |
| Full Contents | v |
| I TCP1_utils | 1 |
| 1 Utility functions | 2 |
| 1.1 Basic utilities | 2 |
| 1.1.1 Summary | 2 |
| 1.1.2 Rules | 2 |
| – <i>funupd</i> | 2 |
| – <i>funupd_list</i> | 2 |
| – <i>clip_int_to_num</i> | 2 |
| – <i>left_shift_num</i> | 2 |
| – <i>right_shift_num</i> | 2 |
| – <i>rounddown</i> | 2 |
| – <i>roundup</i> | 2 |
| – <i>real_of_int</i> | 2 |
| – <i>num_floor</i> | 2 |
| – <i>num_floor_and_frac</i> | 2 |
| – <i>fm_exists</i> | 2 |
| – <i>onlywhen</i> | 2 |
| 1.2 List utilities | 3 |
| 1.2.1 Summary | 3 |
| 1.2.2 Rules | 3 |
| – <i>SPLIT_REV_0</i> | 3 |
| – <i>SPLIT_REV</i> | 3 |
| – <i>SPLIT</i> | 3 |
| – <i>TAKE</i> | 3 |
| – <i>DROP</i> | 3 |
| – <i>TAKEWHILE_REV</i> | 3 |
| – <i>TAKEWHILE</i> | 3 |
| – <i>REPLICATE</i> | 3 |
| – <i>decr_list</i> | 3 |
| – <i>NOTIN'</i> | 3 |
| – <i>MAP_OPTIONAL</i> | 3 |
| – <i>CONCAT_OPTIONAL</i> | 3 |
| – <i>ORDERINGS</i> | 3 |
| – <i>INSERT_ORDERED</i> | 3 |
| 1.3 Assertions | 4 |
| 1.3.1 Summary | 4 |
| 1.3.2 Rules | 4 |
| – <i>ASSERTION_FAILURE</i> | 4 |

| | | |
|------------|--|-----------|
| II | TCP1_errors | 6 |
| 2 | Error codes | 7 |
| 2.1 | The type of errors | 7 |
| 2.1.1 | Summary | 7 |
| 2.1.2 | Rules | 7 |
| – | <i>error</i> | 7 |
| III | TCP1_signals | 9 |
| 3 | Signal names | 10 |
| 3.1 | The type of signals | 10 |
| 3.1.1 | Summary | 10 |
| 3.1.2 | Rules | 10 |
| – | <i>signal</i> | 10 |
| IV | TCP1_baseTypes | 12 |
| 4 | Base types | 13 |
| 4.1 | Network and OS-related types (TCP and UDP) | 13 |
| 4.1.1 | Summary | 13 |
| 4.1.2 | Rules | 13 |
| – | <i>port</i> | 13 |
| – | <i>ip</i> | 13 |
| – | <i>ifid</i> | 13 |
| – | <i>netmask</i> | 14 |
| – | <i>fd</i> | 14 |
| 4.2 | File and socket flags (TCP and UDP) | 14 |
| 4.2.1 | Summary | 14 |
| 4.2.2 | Rules | 14 |
| – | <i>filebflag</i> | 14 |
| – | <i>sockbflag</i> | 14 |
| – | <i>socknflag</i> | 15 |
| – | <i>socktflag</i> | 15 |
| – | <i>msgbflag</i> | 15 |
| – | <i>socktype</i> | 16 |
| 4.3 | Language interaction types | 16 |
| 4.3.1 | Summary | 16 |
| 4.3.2 | Rules | 16 |
| – | <i>tid</i> | 16 |
| – | <i>err</i> | 16 |
| – | <i>TLang_type</i> | 16 |
| – | <i>TLang</i> | 17 |
| – | <i>tlang_typing</i> | 17 |
| 4.4 | Time types | 18 |
| 4.4.1 | Summary | 18 |
| 4.4.2 | Rules | 19 |
| – | <i>time</i> | 19 |
| – | <i>type_abbrev_duration</i> | 19 |
| – | <i>time_lt</i> | 19 |
| – | <i>time_lte</i> | 19 |
| – | <i>time_gt</i> | 19 |
| – | <i>time_gte</i> | 19 |
| – | <i>time_min</i> | 19 |
| – | <i>time_max</i> | 19 |
| – | <i>time_plus_dur</i> | 19 |
| – | <i>time_minus_dur</i> | 19 |

- *real_mult_time* 19
- *time_zero* 20
- *duration* 20
- *abstime* 20
- *realopt_of_time* 20
- *the_time* 20
- 4.5 Basic network types: sequence numbers (TCP only) 20
 - 4.5.1 Summary 21
 - 4.5.2 Rules 21
 - *type_abbrev_byte* 21
 - *seq32* 21
 - *seq32_plus* 21
 - *seq32_minus* 21
 - *seq32_plus'* 21
 - *seq32_minus'* 21
 - *seq32_diff* 21
 - *seq32_lt* 21
 - *seq32_leq* 21
 - *seq32_gt* 21
 - *seq32_geq* 21
 - *seq32_fromto* 21
 - *seq32_coerce* 21
 - *seq32_min* 21
 - *seq32_max* 21
 - *tcpLocal* 22
 - *tcpForeign* 22
 - *type_abbrev_tcp_seq_local* 22
 - *type_abbrev_tcp_seq_foreign* 22
 - *tcp_seq_local* 22
 - *tcp_seq_foreign* 22
 - *tcp_seq_local_to_foreign* 22
 - *tcp_seq_foreign_to_local* 22
 - *tstamp* 22
 - *type_abbrev_ts_seq* 23
 - *ts_seq* 23

V TCP1_netTypes 24

- 5 Network datagram types 25**
 - 5.1 TCP segments (TCP only) 26
 - 5.1.1 Summary 26
 - 5.1.2 Rules 26
 - *tcpSegment* 26
 - *sane_seg* 27
 - 5.2 UDP datagrams (UDP only) 27
 - 5.2.1 Summary 27
 - 5.2.2 Rules 27
 - *udpDatagram* 27
 - *sane_udpdgm* 27
 - 5.3 ICMP datagrams (TCP and UDP) 27
 - 5.3.1 Summary 28
 - 5.3.2 Rules 29
 - *protocol* 29
 - *icmp_unreach_code* 29
 - *icmp_source_quench_code* 29
 - *icmp_redirect_code* 29
 - *icmp_time_exceeded_code* 30
 - *icmp_paramprob_code* 30

| | | |
|-------------|--|-----------|
| – | <i>icmpType</i> | 30 |
| – | <i>icmpDatagram</i> | 30 |
| 5.4 | IP messages (TCP and UDP) | 30 |
| 5.4.1 | Summary | 30 |
| 5.4.2 | Rules | 31 |
| – | <i>msg</i> | 31 |
| – | <i>sane_msg</i> | 31 |
| – | <i>msg_is1</i> | 31 |
| – | <i>msg_is2</i> | 31 |
| VI | TCP1_LIBinterface | 32 |
| 6 | System call types | 33 |
| 6.1 | The interface (TCP and UDP) | 33 |
| 6.1.1 | Summary | 33 |
| 6.1.2 | Rules | 33 |
| – | <i>LIB_interface</i> | 33 |
| – | <i>retType</i> | 34 |
| 6.2 | Useful groups of calls (TCP and UDP) | 35 |
| 6.2.1 | Summary | 35 |
| 6.2.2 | Rules | 35 |
| – | <i>fd_op</i> | 35 |
| – | <i>fd_sockop</i> | 35 |
| VII | TCP1_host0 | 37 |
| 7 | Host LTS labels and rule categories | 38 |
| 7.1 | Transition labels (TCP and UDP) | 38 |
| 7.1.1 | Summary | 38 |
| 7.1.2 | Rules | 38 |
| – | <i>Lhost0</i> | 38 |
| 7.2 | Rule categories (TCP and UDP) | 38 |
| 7.2.1 | Summary | 39 |
| 7.2.2 | Rules | 39 |
| – | <i>rule_proto</i> | 39 |
| – | <i>rule_status</i> | 39 |
| – | <i>rule_cat</i> | 39 |
| – | <i>urgent</i> | 39 |
| – | <i>nonurgent</i> | 39 |
| – | <i>is_urgent</i> | 39 |
| VIII | TCP1_ruleids | 41 |
| 8 | Rule names | 42 |
| 8.1 | names (Rule only) | 42 |
| 8.1.1 | Summary | 42 |
| 8.1.2 | Rules | 42 |
| – | <i>rule_ids</i> | 42 |
| IX | TCP1_timers | 44 |
| 9 | Timers | 45 |
| 9.1 | Properties (TCP and UDP) | 45 |
| 9.1.1 | Summary | 45 |
| 9.1.2 | Rules | 45 |
| – | <i>time_pass_additive</i> | 45 |

- *time_pass_trajectory* 46
- *opttorel* 46
- 9.2 Basic timer *timer* (TCP and UDP) 46
 - 9.2.1 Summary 46
 - 9.2.2 Rules 47
 - *timer* 47
 - *fuzzy_timer* 47
 - *sharp_timer* 47
 - *never_timer* 47
 - *upper_timer* 47
 - *timer_expires* 47
 - *Time_Pass_timer* 47
- 9.3 Deadline timer *timed* (TCP and UDP) 48
 - 9.3.1 Summary 48
 - 9.3.2 Rules 48
 - *timed* 48
 - *timed_val_of* 48
 - *timed_timer_of* 48
 - *timed_expires* 48
 - *Time_Pass_timed* 48
- 9.4 Time-window timer *timewindow* (TCP and UDP) 48
 - 9.4.1 Summary 48
 - 9.4.2 Rules 49
 - *timewindow* 49
 - *timewindow_val_of* 49
 - *timewindow_open* 49
 - *Time_Pass_timewindow* 49
- 9.5 Ticker *ticker* (TCP and UDP) 49
 - 9.5.1 Summary 49
 - 9.5.2 Rules 50
 - *ticker* 50
 - *ticks_of* 50
 - *Time_Pass_ticker* 50
 - *ticker_ok* 50
 - *tick_imin* 50
 - *tick_imax* 50
- 9.6 Stopwatch *stopwatch* (TCP and UDP) 50
 - 9.6.1 Summary 50
 - 9.6.2 Rules 51
 - *stopwatch* 51
 - *stopwatch_val_of* 51
 - *Time_Pass_stopwatch* 51

X TCP1_hostTypes 52

10 Host types 53

- 10.1 Files (TCP and UDP) 53
 - 10.1.1 Summary 53
 - 10.1.2 Rules 53
 - *fid* 53
 - *sid* 53
 - *filetype* 53
 - *fileflags* 53
 - *file* 53
 - *File* 53
- 10.2 TCP states (TCP only) 54
 - 10.2.1 Summary 54
 - 10.2.2 Rules 54

| | | |
|--------|--|----|
| – | <i>tcpstate</i> | 54 |
| 10.3 | The TCP control block (TCP only) | 54 |
| 10.3.1 | Summary | 54 |
| 10.3.2 | Rules | 54 |
| – | <i>tcpReassSegment</i> | 54 |
| – | <i>rexmtmode</i> | 55 |
| – | <i>rttinf</i> | 55 |
| – | <i>tcpcb</i> | 55 |
| 10.4 | Sockets (TCP and UDP) | 57 |
| 10.4.1 | Summary | 57 |
| 10.4.2 | Rules | 57 |
| – | <i>iobc</i> | 57 |
| – | <i>socket_listen</i> | 57 |
| – | <i>tcp_socket</i> | 58 |
| – | <i>dgram_msg</i> | 58 |
| – | <i>dgram_error</i> | 58 |
| – | <i>dgram</i> | 58 |
| – | <i>udp_socket</i> | 58 |
| – | <i>sockflags</i> | 58 |
| – | <i>protocol_info</i> | 58 |
| – | <i>socket</i> | 59 |
| – | <i>TCP_Sock0</i> | 59 |
| – | <i>TCP_Sock</i> | 59 |
| – | <i>UDP_Sock0</i> | 59 |
| – | <i>UDP_Sock</i> | 59 |
| – | <i>Sock</i> | 59 |
| – | <i>tcp_sock_of</i> | 59 |
| – | <i>udp_sock_of</i> | 59 |
| – | <i>proto_of</i> | 59 |
| – | <i>proto_eq</i> | 59 |
| 10.5 | The host (TCP and UDP) | 60 |
| 10.5.1 | Summary | 60 |
| 10.5.2 | Rules | 60 |
| – | <i>arch</i> | 60 |
| – | <i>ifd</i> | 60 |
| – | <i>routing_table_entry</i> | 60 |
| – | <i>type_abbrev_routing_table</i> | 61 |
| – | <i>bandlim_reason</i> | 61 |
| – | <i>type_abbrev_bandlim_state</i> | 61 |
| – | <i>hostThreadState</i> | 61 |
| – | <i>host</i> | 61 |
| 10.6 | Trace records (TCP and UDP) | 62 |
| 10.6.1 | Summary | 62 |
| 10.6.2 | Rules | 62 |
| – | <i>traceflavour</i> | 62 |
| – | <i>type_abbrev_tracerecord</i> | 62 |
| – | <i>tracecb_eq</i> | 62 |
| – | <i>tracesock_eq</i> | 63 |

XI TCP1_params 65

11 Host behavioural parameters 66

| | | |
|--------|---|----|
| 11.1 | Model parameters (TCP and UDP) | 66 |
| 11.1.1 | Summary | 66 |
| 11.1.2 | Rules | 66 |
| – | <i>INFINITE_RESOURCES</i> | 66 |
| – | <i>BSD_RTTVAR_BUG</i> | 66 |
| 11.2 | Scheduling parameters (TCP and UDP) | 67 |

| | | |
|--------|---|----|
| 11.2.1 | Summary | 67 |
| 11.2.2 | Rules | 67 |
| – | <i>dschedmax</i> | 67 |
| – | <i>dqmax</i> | 67 |
| – | <i>doqmax</i> | 67 |
| 11.3 | Timers (TCP and UDP) | 67 |
| 11.3.1 | Summary | 67 |
| 11.3.2 | Rules | 67 |
| – | <i>HZ</i> | 68 |
| – | <i>tickintvlmin</i> | 68 |
| – | <i>tickintvlmax</i> | 68 |
| – | <i>stopwatchfuzz</i> | 68 |
| – | <i>stopwatch_zero</i> | 68 |
| – | <i>SLOW_TIMER_INTVL</i> | 68 |
| – | <i>SLOW_TIMER_MODEL_INTVL</i> | 68 |
| – | <i>FAST_TIMER_INTVL</i> | 68 |
| – | <i>FAST_TIMER_MODEL_INTVL</i> | 68 |
| – | <i>KERN_TIMER_INTVL</i> | 68 |
| – | <i>KERN_TIMER_MODEL_INTVL</i> | 68 |
| 11.4 | Ports, sockets, and files (TCP and UDP) | 69 |
| 11.4.1 | Summary | 69 |
| 11.4.2 | Rules | 69 |
| – | <i>privileged_ports</i> | 69 |
| – | <i>ephemeral_ports</i> | 69 |
| – | <i>OPEN_MAX</i> | 69 |
| – | <i>OPEN_MAX_FD</i> | 69 |
| – | <i>FD_SETSIZE</i> | 69 |
| – | <i>SOMAXCONN</i> | 70 |
| 11.5 | UDP parameters (UDP only) | 70 |
| 11.5.1 | Summary | 70 |
| 11.5.2 | Rules | 70 |
| – | <i>UDPpayloadMax</i> | 70 |
| 11.6 | Buffers (TCP and UDP) | 70 |
| 11.6.1 | Summary | 70 |
| 11.6.2 | Rules | 70 |
| – | <i>MCLBYTES</i> | 70 |
| – | <i>MSIZE</i> | 70 |
| – | <i>SB_MAX</i> | 70 |
| – | <i>oob_extra_sndbuf</i> | 70 |
| 11.7 | File and socket flag defaults (TCP and UDP) | 71 |
| 11.7.1 | Summary | 71 |
| 11.7.2 | Rules | 71 |
| – | <i>ff_default_b</i> | 71 |
| – | <i>ff_default</i> | 71 |
| – | <i>sf_default_b</i> | 71 |
| – | <i>sf_default_n</i> | 71 |
| – | <i>sf_default_t</i> | 72 |
| – | <i>sf_default</i> | 72 |
| – | <i>sf_min_n</i> | 72 |
| – | <i>sf_max_n</i> | 72 |
| – | <i>sndrcv_timeo_t_max</i> | 73 |
| – | <i>pselect_timeo_t_max</i> | 73 |
| 11.8 | RFC-specified limits (TCP only) | 73 |
| 11.8.1 | Summary | 73 |
| 11.8.2 | Rules | 73 |
| – | <i>dtsinval</i> | 73 |
| – | <i>TCP_MAXWIN</i> | 73 |
| – | <i>TCP_MAXWINSCALE</i> | 73 |

| | | |
|---------|--------------------------------------|----|
| 11.9 | Protocol parameters (TCP only) | 74 |
| 11.9.1 | Summary | 74 |
| 11.9.2 | Rules | 74 |
| – | <i>MSSDFLT</i> | 74 |
| – | <i>SS_FLTSZ_LOCAL</i> | 74 |
| – | <i>SS_FLTSZ</i> | 74 |
| – | <i>TCP_DO_NEWRENO</i> | 74 |
| – | <i>TCP_Q0MINLIMIT</i> | 74 |
| – | <i>TCP_Q0MAXLIMIT</i> | 74 |
| – | <i>backlog_fudge</i> | 75 |
| 11.10 | Time values (TCP only) | 75 |
| 11.10.1 | Summary | 75 |
| 11.10.2 | Rules | 75 |
| – | <i>TCPTV_DELACK</i> | 75 |
| – | <i>TCPTV_RTOBASE</i> | 75 |
| – | <i>TCPTV_RTTVARBASE</i> | 75 |
| – | <i>TCPTV_MIN</i> | 75 |
| – | <i>TCPTV_REXMTMAX</i> | 75 |
| – | <i>TCPTV_MSL</i> | 76 |
| – | <i>TCPTV_PERSMIN</i> | 76 |
| – | <i>TCPTV_PERSMAX</i> | 76 |
| – | <i>TCPTV_KEEP_INIT</i> | 76 |
| – | <i>TCPTV_KEEP_IDLE</i> | 76 |
| – | <i>TCPTV_KEEPINTVL</i> | 76 |
| – | <i>TCPTV_KEEPCNT</i> | 76 |
| – | <i>TCPTV_MAXIDLE</i> | 76 |
| 11.11 | Timing-related parameters (TCP only) | 76 |
| 11.11.1 | Summary | 76 |
| 11.11.2 | Rules | 76 |
| – | <i>TCP_BSD_BACKOFFS</i> | 76 |
| – | <i>TCP_LINUX_BACKOFFS</i> | 76 |
| – | <i>TCP_WINXP_BACKOFFS</i> | 76 |
| – | <i>TCP_MAXRXTSHIFT</i> | 77 |
| – | <i>TCP_SYNACKMAXRXTSHIFT</i> | 77 |
| – | <i>TCP_SYN_BSD_BACKOFFS</i> | 77 |
| – | <i>TCP_SYN_LINUX_BACKOFFS</i> | 77 |
| – | <i>TCP_SYN_WINXP_BACKOFFS</i> | 77 |

XII TCP1_auxFns 78

| | | |
|-----------|---|-----------|
| 12 | Auxiliary functions | 79 |
| 12.1 | Architecture handling (TCP and UDP) | 79 |
| 12.1.1 | Summary | 79 |
| 12.1.2 | Rules | 79 |
| – | <i>windows_arch</i> | 79 |
| – | <i>bsd_arch</i> | 79 |
| – | <i>linux_arch</i> | 79 |
| – | <i>unix_arch</i> | 79 |
| 12.2 | Interfaces and IP addresses (TCP and UDP) | 79 |
| 12.2.1 | Summary | 79 |
| 12.2.2 | Rules | 80 |
| – | <i>mask</i> | 80 |
| – | <i>mask_bits</i> | 80 |
| – | <i>IP</i> | 80 |
| – | <i>IN_MULTICAST</i> | 80 |
| – | <i>INADDR_BROADCAST</i> | 80 |
| – | <i>LOOPBACK_ADDRS</i> | 80 |
| – | <i>ip_localhost</i> | 80 |

- *in_loopback* 80
- *in_local* 80
- *local_ips* 80
- *local_primary_ips* 80
- *is_localnet* 80
- *if_broadcast* 80
- *if_any* 80
- *is_broadormulticast* 81
- *routeable* 81
- *outroute_ifids* 81
- *ifid_up* 82
- *outroute* 82
- *auto_outroute* 82
- *test_outroute_ip* 82
- *test_outroute* 82
- *loopback_on_wire* 83
- 12.3 Files, file descriptors, and sockets (TCP and UDP) 83
 - 12.3.1 Summary 83
 - 12.3.2 Rules 83
 - *fdlt* 83
 - *fdle* 83
 - *leastfd* 83
 - *nextfd* 83
 - *fid_ref_count* 84
 - *sane_socket* 84
- 12.4 Binding (TCP and UDP) 84
 - 12.4.1 Summary 84
 - 12.4.2 Rules 85
 - *bound_ports_protocol_autobind* 85
 - *bound_port_allowed* 85
 - *autobind* 85
 - *bound_after* 85
 - *match_score* 85
 - *lookup_udp* 86
 - *tcp_socket_best_match* 86
 - *lookup_icmp* 87
- 12.5 Timers (TCP and UDP) 88
 - 12.5.1 Summary 88
 - 12.5.2 Rules 88
 - *slow_timer* 88
 - *fast_timer* 88
 - *kern_timer* 88
 - *sched_timer* 88
 - *inqueue_timer* 88
 - *outqueue_timer* 88
- 12.6 Time values for socket options (TCP and UDP) 89
 - 12.6.1 Summary 89
 - 12.6.2 Rules 89
 - *time_of_tlltime* 89
 - *time_of_tlltimeopt* 89
 - *tlltimeopt_wf* 89
 - *tlltimeopt_of_time* 89
- 12.7 Queues (TCP and UDP) 89
 - 12.7.1 Summary 90
 - 12.7.2 Rules 90
 - *enqueue* 90
 - *enqueue_iq* 90
 - *enqueue_oq* 90

| | | |
|---------|--|-----|
| - | <i>dequeue</i> | 90 |
| - | <i>dequeue_iq</i> | 90 |
| - | <i>dequeue_oq</i> | 90 |
| - | <i>route_and_enqueue_oq</i> | 91 |
| - | <i>enqueue_list_qinfo</i> | 91 |
| - | <i>enqueue_list</i> | 91 |
| - | <i>enqueue_oq_list_qinfo</i> | 91 |
| - | <i>enqueue_oq_list</i> | 91 |
| - | <i>accept_incoming_q0</i> | 91 |
| - | <i>accept_incoming_q</i> | 91 |
| - | <i>drop_from_q0</i> | 91 |
| 12.8 | TCP Options (TCP only) | 92 |
| 12.8.1 | Summary | 92 |
| 12.8.2 | Rules | 92 |
| - | <i>do_tcp_options</i> | 92 |
| - | <i>calculate_tcp_options_len</i> | 92 |
| 12.9 | Buffers, windows, and queues (TCP and UDP) | 92 |
| 12.9.1 | Summary | 92 |
| 12.9.2 | Rules | 93 |
| - | <i>calculate_buf_sizes</i> | 93 |
| - | <i>calculate_bsd_rcv_wnd</i> | 93 |
| - | <i>send_queue_space</i> | 93 |
| 12.10 | Band limiting (TCP and UDP) | 94 |
| 12.10.1 | Summary | 94 |
| 12.10.2 | Rules | 94 |
| - | <i>bandlim_state_init</i> | 94 |
| - | <i>bandlim_rst_ok_always</i> | 94 |
| - | <i>simple_limit</i> | 94 |
| - | <i>bandlim_rst_ok_simple</i> | 94 |
| - | <i>bandlim_rst_ok</i> | 95 |
| - | <i>enqueue_oq_bndlim_rst</i> | 95 |
| 12.11 | UDP support (UDP only) | 95 |
| 12.11.1 | Summary | 95 |
| 12.11.2 | Rules | 95 |
| - | <i>dosend</i> | 96 |
| 12.12 | TCP timing and RTT (TCP only) | 96 |
| 12.12.1 | Summary | 96 |
| 12.12.2 | Rules | 96 |
| - | <i>tcp_backoffs</i> | 96 |
| - | <i>tcp_syn_backoffs</i> | 96 |
| - | <i>mode_of</i> | 97 |
| - | <i>shift_of</i> | 97 |
| - | <i>computed_rto</i> | 97 |
| - | <i>computed_rxtcur</i> | 97 |
| - | <i>start_tt_rexmt_gen</i> | 97 |
| - | <i>start_tt_rexmt</i> | 97 |
| - | <i>start_tt_rexmtsyn</i> | 97 |
| - | <i>start_tt_persist</i> | 97 |
| - | <i>update_rtt</i> | 98 |
| - | <i>expand_cwnd</i> | 99 |
| 12.13 | Path MTU Discovery (TCP only) | 99 |
| 12.13.1 | Summary | 99 |
| 12.13.2 | Rules | 99 |
| - | <i>next_smaller</i> | 99 |
| - | <i>mtu_tab</i> | 99 |
| 12.14 | Reassembly (TCP only) | 100 |
| 12.14.1 | Summary | 100 |
| 12.14.2 | Rules | 100 |

| | | |
|-----------|--|------------|
| – | <i>tcp_reass</i> | 100 |
| – | <i>tcp_reass_prune</i> | 101 |
| 12.15 | The initial TCP control block (TCP only) | 101 |
| 12.15.1 | Summary | 101 |
| 12.15.2 | Rules | 101 |
| – | <i>initial_cb</i> | 101 |
| 13 | Relational monad | 103 |
| 13.1 | Relational monad (TCP only) | 103 |
| 13.1.1 | Summary | 103 |
| 13.1.2 | Rules | 104 |
| – | <i>andThen</i> | 104 |
| – | <i>cont</i> | 104 |
| – | <i>stop</i> | 104 |
| – | <i>assert</i> | 104 |
| – | <i>assert_failure</i> | 104 |
| – | <i>chooseM</i> | 104 |
| – | <i>get_sock</i> | 104 |
| – | <i>get_tcp_sock</i> | 104 |
| – | <i>get_cb</i> | 104 |
| – | <i>modify_sock</i> | 104 |
| – | <i>modify_tcp_sock</i> | 104 |
| – | <i>modify_cb</i> | 104 |
| – | <i>emit_segs</i> | 105 |
| – | <i>emit_segs_pred</i> | 105 |
| – | <i>miftc</i> | 105 |
| – | <i>miftc_bndlm</i> | 105 |
| 14 | Auxiliary functions for TCP segment creation and drop | 106 |
| 14.1 | SYN and RST Segment Creation (TCP only) | 106 |
| 14.1.1 | Summary | 106 |
| 14.1.2 | Rules | 106 |
| – | <i>make_syn_segment</i> | 106 |
| – | <i>make_syn_ack_segment</i> | 107 |
| – | <i>make_ack_segment</i> | 108 |
| – | <i>bsd_make_phantom_segment</i> | 109 |
| – | <i>make_rst_segment_from_cb</i> | 109 |
| – | <i>make_rst_segment_from_seg</i> | 110 |
| 14.2 | General Segment Creation (TCP only) | 111 |
| 14.2.1 | Summary | 111 |
| 14.2.2 | Rules | 111 |
| – | <i>tcp_output_required</i> | 111 |
| – | <i>tcp_output_really</i> | 113 |
| – | <i>tcp_output_perhaps</i> | 116 |
| 14.3 | Segment Queueing (TCP only) | 116 |
| 14.3.1 | Summary | 116 |
| 14.3.2 | Rules | 117 |
| – | <i>rollback_tcp_output</i> | 117 |
| – | <i>enqueue_or_fail</i> | 118 |
| – | <i>enqueue_or_fail_sock</i> | 118 |
| – | <i>enqueue_and_ignore_fail</i> | 118 |
| – | <i>enqueue_each_and_ignore_fail</i> | 118 |
| – | <i>mift_tcp_output_perhaps_or_fail</i> | 118 |
| 14.4 | Incoming Segment Functions (TCP only) | 119 |
| 14.4.1 | Summary | 119 |
| 14.4.2 | Rules | 119 |
| – | <i>update_idle</i> | 119 |
| 14.5 | Drop Segment Functions (TCP only) | 119 |
| 14.5.1 | Summary | 119 |

| | | |
|-------------|--------------------------------------|------------|
| 14.5.2 | Rules | 120 |
| – | <i>dropwithreset</i> | 120 |
| – | <i>mlipt_dropafterack_or_fail</i> | 120 |
| – | <i>dropwithreset_ignore_fail</i> | 120 |
| 14.6 | Close Functions (TCP only) | 121 |
| 14.6.1 | Summary | 121 |
| 14.6.2 | Rules | 121 |
| – | <i>tcp_close</i> | 121 |
| – | <i>tcp_drop_and_close</i> | 121 |
| XIII | TCP1_hostLTS | 123 |
| 15 | Host LTS: Socket Calls | 124 |
| 15.1 | <code>accept()</code> (TCP only) | 124 |
| 15.1.1 | Errors | 124 |
| 15.1.2 | Common cases | 125 |
| 15.1.3 | API | 125 |
| 15.1.4 | Model details | 125 |
| 15.1.5 | Summary | 125 |
| 15.1.6 | Rules | 126 |
| – | <i>accept_1</i> | 126 |
| – | <i>accept_2</i> | 127 |
| – | <i>accept_3</i> | 127 |
| – | <i>accept_4</i> | 128 |
| – | <i>accept_5</i> | 129 |
| – | <i>accept_6</i> | 129 |
| – | <i>accept_7</i> | 130 |
| 15.2 | <code>bind()</code> (TCP and UDP) | 130 |
| 15.2.1 | Errors | 131 |
| 15.2.2 | Common cases | 131 |
| 15.2.3 | API | 131 |
| 15.2.4 | Model details | 132 |
| 15.2.5 | Summary | 132 |
| 15.2.6 | Rules | 133 |
| – | <i>bind_1</i> | 133 |
| – | <i>bind_2</i> | 134 |
| – | <i>bind_3</i> | 134 |
| – | <i>bind_5</i> | 135 |
| – | <i>bind_7</i> | 135 |
| – | <i>bind_9</i> | 135 |
| 15.3 | <code>close()</code> (TCP and UDP) | 136 |
| 15.3.1 | Errors | 137 |
| 15.3.2 | Common cases | 137 |
| 15.3.3 | API | 137 |
| 15.3.4 | Model details | 137 |
| 15.3.5 | Summary | 137 |
| 15.3.6 | Rules | 138 |
| – | <i>close_1</i> | 138 |
| – | <i>close_2</i> | 138 |
| – | <i>close_3</i> | 139 |
| – | <i>close_4</i> | 140 |
| – | <i>close_5</i> | 141 |
| – | <i>close_6</i> | 142 |
| – | <i>close_7</i> | 142 |
| – | <i>close_8</i> | 143 |
| – | <i>close_10</i> | 144 |
| 15.4 | <code>connect()</code> (TCP and UDP) | 145 |
| 15.4.1 | Errors | 146 |

| | | |
|--------|-------------------------------------|-----|
| 15.4.2 | Common cases | 147 |
| 15.4.3 | API | 147 |
| 15.4.4 | Model details | 147 |
| 15.4.5 | Summary | 148 |
| 15.4.6 | Rules | 148 |
| – | <i>connect_1</i> | 148 |
| – | <i>connect_2</i> | 152 |
| – | <i>connect_3</i> | 152 |
| – | <i>connect_4</i> | 153 |
| – | <i>connect_4a</i> | 154 |
| – | <i>connect_5</i> | 154 |
| – | <i>connect_5a</i> | 155 |
| – | <i>connect_5b</i> | 156 |
| – | <i>connect_5c</i> | 157 |
| – | <i>connect_5d</i> | 157 |
| – | <i>connect_6</i> | 158 |
| – | <i>connect_7</i> | 158 |
| – | <i>connect_8</i> | 159 |
| – | <i>connect_9</i> | 160 |
| – | <i>connect_10</i> | 161 |
| 15.5 | disconnect() (TCP and UDP) | 161 |
| 15.5.1 | Errors | 162 |
| 15.5.2 | Common cases | 162 |
| 15.5.3 | API | 162 |
| 15.5.4 | Summary | 163 |
| 15.5.5 | Rules | 163 |
| – | <i>disconnect_4</i> | 163 |
| – | <i>disconnect_5</i> | 164 |
| – | <i>disconnect_1</i> | 164 |
| – | <i>disconnect_2</i> | 165 |
| – | <i>disconnect_3</i> | 166 |
| 15.6 | dup() (TCP and UDP) | 166 |
| 15.6.1 | Errors | 166 |
| 15.6.2 | Common cases | 167 |
| 15.6.3 | API | 167 |
| 15.6.4 | Summary | 167 |
| 15.6.5 | Rules | 167 |
| – | <i>dup_1</i> | 167 |
| – | <i>dup_2</i> | 167 |
| 15.7 | dupfd() (TCP and UDP) | 168 |
| 15.7.1 | Errors | 168 |
| 15.7.2 | Common cases | 168 |
| 15.7.3 | API | 168 |
| 15.7.4 | Model details | 169 |
| 15.7.5 | Summary | 169 |
| 15.7.6 | Rules | 169 |
| – | <i>dupfd_1</i> | 169 |
| – | <i>dupfd_3</i> | 170 |
| – | <i>dupfd_4</i> | 170 |
| 15.8 | getfileflags() (TCP and UDP) | 170 |
| 15.8.1 | Errors | 171 |
| 15.8.2 | Common cases | 171 |
| 15.8.3 | API | 171 |
| 15.8.4 | Model details | 171 |
| 15.8.5 | Summary | 171 |
| 15.8.6 | Rules | 171 |
| – | <i>getfileflags_1</i> | 171 |
| 15.9 | getifaddrs() (TCP and UDP) | 172 |

| | | |
|---------|---|-----|
| 15.9.1 | Errors | 172 |
| 15.9.2 | Common cases | 172 |
| 15.9.3 | API | 172 |
| 15.9.4 | Model details | 173 |
| 15.9.5 | Summary | 173 |
| 15.9.6 | Rules | 173 |
| – | <i>getifaddrs_1</i> | 173 |
| 15.10 | getpeername() (TCP and UDP) | 173 |
| 15.10.1 | Errors | 174 |
| 15.10.2 | Common cases | 174 |
| 15.10.3 | API | 174 |
| 15.10.4 | Model details | 174 |
| 15.10.5 | Summary | 175 |
| 15.10.6 | Rules | 175 |
| – | <i>getpeername_1</i> | 175 |
| – | <i>getpeername_2</i> | 176 |
| 15.11 | getsockbopt() (TCP and UDP) | 176 |
| 15.11.1 | Errors | 177 |
| 15.11.2 | Common cases | 177 |
| 15.11.3 | API | 177 |
| 15.11.4 | Model details | 177 |
| 15.11.5 | Summary | 178 |
| 15.11.6 | Rules | 178 |
| – | <i>getsockbopt_1</i> | 178 |
| – | <i>getsockbopt_2</i> | 178 |
| 15.12 | getsockerr() (TCP and UDP) | 179 |
| 15.12.1 | Errors | 179 |
| 15.12.2 | Common cases | 179 |
| 15.12.3 | API | 179 |
| 15.12.4 | Model details | 180 |
| 15.12.5 | Summary | 180 |
| 15.12.6 | Rules | 180 |
| – | <i>getsockerr_1</i> | 180 |
| – | <i>getsockerr_2</i> | 180 |
| 15.13 | getsocklistening() (TCP and UDP) | 181 |
| 15.13.1 | Errors | 181 |
| 15.13.2 | Common cases | 181 |
| 15.13.3 | API | 181 |
| 15.13.4 | Model details | 182 |
| 15.13.5 | Summary | 182 |
| 15.13.6 | Rules | 182 |
| – | <i>getsocklistening_1</i> | 182 |
| – | <i>getsocklistening_3</i> | 182 |
| – | <i>getsocklistening_2</i> | 183 |
| 15.14 | getsockname() (TCP and UDP) | 183 |
| 15.14.1 | Errors | 184 |
| 15.14.2 | Common cases | 184 |
| 15.14.3 | API | 184 |
| 15.14.4 | Model details | 184 |
| 15.14.5 | Summary | 185 |
| 15.14.6 | Rules | 185 |
| – | <i>getsockname_1</i> | 185 |
| – | <i>getsockname_2</i> | 185 |
| – | <i>getsockname_3</i> | 186 |
| 15.15 | getsocknopt() (TCP and UDP) | 187 |
| 15.15.1 | Errors | 187 |
| 15.15.2 | Common cases | 187 |
| 15.15.3 | API | 187 |

| | | |
|---------|---|-----|
| 15.15.4 | Model details | 188 |
| 15.15.5 | Summary | 188 |
| 15.15.6 | Rules | 188 |
| – | <i>getsocknopt_1</i> | 188 |
| – | <i>getsocknopt_4</i> | 188 |
| 15.16 | getsockopt() (TCP and UDP) | 189 |
| 15.16.1 | Errors | 189 |
| 15.16.2 | Common cases | 189 |
| 15.16.3 | API | 189 |
| 15.16.4 | Model details | 190 |
| 15.16.5 | Summary | 190 |
| 15.16.6 | Rules | 190 |
| – | <i>getsockopt_1</i> | 190 |
| – | <i>getsockopt_4</i> | 191 |
| 15.17 | listen() (TCP only) | 191 |
| 15.17.1 | Errors | 192 |
| 15.17.2 | Common cases | 192 |
| 15.17.3 | API | 192 |
| 15.17.4 | Model details | 192 |
| 15.17.5 | Summary | 193 |
| 15.17.6 | Rules | 193 |
| – | <i>listen_1</i> | 193 |
| – | <i>listen_1b</i> | 194 |
| – | <i>listen_1c</i> | 194 |
| – | <i>listen_2</i> | 195 |
| – | <i>listen_3</i> | 195 |
| – | <i>listen_4</i> | 196 |
| – | <i>listen_5</i> | 197 |
| – | <i>listen_7</i> | 197 |
| 15.18 | pselect() (TCP and UDP) | 198 |
| 15.18.1 | Errors | 198 |
| 15.18.2 | Common cases | 198 |
| 15.18.3 | API | 199 |
| 15.18.4 | Model details | 200 |
| 15.18.5 | Summary | 200 |
| 15.18.6 | Rules | 200 |
| – | <i>pselect_1</i> | 200 |
| – | <i>soreadable</i> | 202 |
| – | <i>sowriteable</i> | 202 |
| – | <i>soexceptional</i> | 203 |
| – | <i>pselect_2</i> | 203 |
| – | <i>pselect_3</i> | 203 |
| – | <i>pselect_4</i> | 204 |
| – | <i>pselect_5</i> | 205 |
| – | <i>pselect_6</i> | 205 |
| 15.19 | recv() (TCP only) | 206 |
| 15.19.1 | Errors | 207 |
| 15.19.2 | Common cases | 208 |
| 15.19.3 | API | 208 |
| 15.19.4 | Model details | 209 |
| 15.19.5 | Summary | 209 |
| 15.19.6 | Rules | 209 |
| – | <i>recv_1</i> | 209 |
| – | <i>recv_2</i> | 211 |
| – | <i>recv_3</i> | 211 |
| – | <i>recv_4</i> | 213 |
| – | <i>recv_5</i> | 214 |
| – | <i>recv_6</i> | 214 |

| | | |
|---------|--------------------------|-----|
| – | <i>recv_7</i> | 215 |
| – | <i>recv_8</i> | 215 |
| – | <i>recv_8a</i> | 216 |
| – | <i>recv_9</i> | 217 |
| 15.20 | <i>recv()</i> (UDP only) | 218 |
| 15.20.1 | Errors | 218 |
| 15.20.2 | Common cases | 219 |
| 15.20.3 | API | 219 |
| 15.20.4 | Model details | 220 |
| 15.20.5 | Summary | 220 |
| 15.20.6 | Rules | 221 |
| – | <i>recv_11</i> | 221 |
| – | <i>recv_12</i> | 222 |
| – | <i>recv_13</i> | 222 |
| – | <i>recv_14</i> | 223 |
| – | <i>recv_15</i> | 224 |
| – | <i>recv_16</i> | 224 |
| – | <i>recv_17</i> | 225 |
| – | <i>recv_20</i> | 225 |
| – | <i>recv_21</i> | 227 |
| – | <i>recv_22</i> | 227 |
| – | <i>recv_23</i> | 228 |
| – | <i>recv_24</i> | 228 |
| 15.21 | <i>send()</i> (TCP only) | 229 |
| 15.21.1 | Errors | 229 |
| 15.21.2 | Common cases | 230 |
| 15.21.3 | API | 230 |
| 15.21.4 | Model details | 230 |
| 15.21.5 | Summary | 231 |
| 15.21.6 | Rules | 231 |
| – | <i>send_1</i> | 231 |
| – | <i>send_2</i> | 234 |
| – | <i>send_3</i> | 235 |
| – | <i>send_3a</i> | 235 |
| – | <i>send_4</i> | 236 |
| – | <i>send_5</i> | 237 |
| – | <i>send_5a</i> | 237 |
| – | <i>send_6</i> | 237 |
| – | <i>send_7</i> | 238 |
| – | <i>send_8</i> | 239 |
| 15.22 | <i>send()</i> (UDP only) | 239 |
| 15.22.1 | Errors | 240 |
| 15.22.2 | Common cases | 241 |
| 15.22.3 | API | 241 |
| 15.22.4 | Model details | 241 |
| 15.22.5 | Summary | 242 |
| 15.22.6 | Rules | 243 |
| – | <i>send_9</i> | 243 |
| – | <i>send_10</i> | 244 |
| – | <i>send_11</i> | 245 |
| – | <i>send_12</i> | 246 |
| – | <i>send_13</i> | 247 |
| – | <i>send_14</i> | 247 |
| – | <i>send_15</i> | 248 |
| – | <i>send_16</i> | 249 |
| – | <i>send_17</i> | 249 |
| – | <i>send_18</i> | 250 |
| – | <i>send_19</i> | 250 |

| | | |
|---------|-------------------------------------|-----|
| – | <i>send_21</i> | 251 |
| – | <i>send_22</i> | 252 |
| – | <i>send_23</i> | 253 |
| 15.23 | setfileflags() (TCP and UDP) | 253 |
| 15.23.1 | Errors | 253 |
| 15.23.2 | Common cases | 254 |
| 15.23.3 | API | 254 |
| 15.23.4 | Model details | 254 |
| 15.23.5 | Summary | 254 |
| 15.23.6 | Rules | 254 |
| – | <i>setfileflags_1</i> | 254 |
| 15.24 | setsockbopt() (TCP and UDP) | 255 |
| 15.24.1 | Errors | 255 |
| 15.24.2 | Common cases | 255 |
| 15.24.3 | API | 255 |
| 15.24.4 | Model details | 256 |
| 15.24.5 | Summary | 256 |
| 15.24.6 | Rules | 256 |
| – | <i>setsockbopt_1</i> | 256 |
| – | <i>setsockbopt_2</i> | 257 |
| 15.25 | setsocknopt() (TCP and UDP) | 257 |
| 15.25.1 | Errors | 258 |
| 15.25.2 | Common cases | 258 |
| 15.25.3 | API | 258 |
| 15.25.4 | Model details | 258 |
| 15.25.5 | Summary | 259 |
| 15.25.6 | Rules | 259 |
| – | <i>setsocknopt_1</i> | 259 |
| – | <i>setsocknopt_2</i> | 259 |
| – | <i>setsocknopt_4</i> | 260 |
| 15.26 | setsocktopt() (TCP and UDP) | 260 |
| 15.26.1 | Errors | 261 |
| 15.26.2 | Common cases | 261 |
| 15.26.3 | API | 261 |
| 15.26.4 | Model details | 262 |
| 15.26.5 | Summary | 262 |
| 15.26.6 | Rules | 262 |
| – | <i>setsocktopt_1</i> | 262 |
| – | <i>setsocktopt_4</i> | 262 |
| – | <i>setsocktopt_5</i> | 263 |
| 15.27 | shutdown() (TCP and UDP) | 263 |
| 15.27.1 | Errors | 264 |
| 15.27.2 | Common cases | 264 |
| 15.27.3 | API | 264 |
| 15.27.4 | Model details | 264 |
| 15.27.5 | Summary | 265 |
| 15.27.6 | Rules | 265 |
| – | <i>shutdown_1</i> | 265 |
| – | <i>shutdown_2</i> | 266 |
| – | <i>shutdown_3</i> | 266 |
| – | <i>shutdown_4</i> | 267 |
| 15.28 | socketmark() (TCP only) | 267 |
| 15.28.1 | Errors | 268 |
| 15.28.2 | Common cases | 268 |
| 15.28.3 | API | 268 |
| 15.28.4 | Model details | 268 |
| 15.28.5 | Summary | 269 |
| 15.28.6 | Rules | 269 |

| | | |
|-----------|---------------------------------------|------------|
| – | <i>socketmark_1</i> | 269 |
| – | <i>socketmark_2</i> | 269 |
| 15.29 | socket() (TCP and UDP) | 271 |
| 15.29.1 | Errors | 271 |
| 15.29.2 | Common cases | 271 |
| 15.29.3 | API | 271 |
| 15.29.4 | Model details | 272 |
| 15.29.5 | Summary | 272 |
| 15.29.6 | Rules | 272 |
| – | <i>socket_1</i> | 272 |
| – | <i>socket_2</i> | 273 |
| 15.30 | Miscellaneous (TCP and UDP) | 273 |
| 15.30.1 | Errors | 273 |
| 15.30.2 | Summary | 274 |
| 15.30.3 | Rules | 274 |
| – | <i>return_1</i> | 274 |
| – | <i>badf_1</i> | 274 |
| – | <i>notsock_1</i> | 275 |
| – | <i>intr_1</i> | 275 |
| – | <i>resourcefail_1</i> | 276 |
| – | <i>resourcefail_2</i> | 276 |
| 16 | Host LTS: TCP Input Processing | 278 |
| 16.1 | Input Processing (TCP only) | 278 |
| 16.1.1 | Summary | 278 |
| 16.1.2 | Rules | 279 |
| – | <i>deliver_in_1</i> | 279 |
| – | <i>deliver_in_1b</i> | 283 |
| – | <i>deliver_in_2</i> | 285 |
| – | <i>deliver_in_2a</i> | 290 |
| – | <i>deliver_in_3</i> | 291 |
| – | <i>di3_topstuff</i> | 294 |
| – | <i>di3_newackstuff</i> | 295 |
| – | <i>di3_ackstuff</i> | 298 |
| – | <i>di3_datastuff_really</i> | 300 |
| – | <i>di3_datastuff</i> | 304 |
| – | <i>di3_ststuff</i> | 305 |
| – | <i>di3_socks_update</i> | 308 |
| – | <i>deliver_in_3a</i> | 309 |
| – | <i>deliver_in_3b</i> | 310 |
| – | <i>deliver_in_3c</i> | 311 |
| – | <i>deliver_in_4</i> | 312 |
| – | <i>deliver_in_5</i> | 313 |
| – | <i>deliver_in_6</i> | 313 |
| – | <i>deliver_in_7</i> | 314 |
| – | <i>deliver_in_7a</i> | 315 |
| – | <i>deliver_in_7b</i> | 316 |
| – | <i>deliver_in_7c</i> | 317 |
| – | <i>deliver_in_7d</i> | 318 |
| – | <i>deliver_in_8</i> | 319 |
| – | <i>deliver_in_9</i> | 320 |
| 17 | Host LTS: TCP Output | 322 |
| 17.1 | Output (TCP only) | 322 |
| 17.1.1 | Summary | 323 |
| 17.1.2 | Rules | 323 |
| – | <i>deliver_out_1</i> | 323 |

| | |
|---|------------|
| 18 Host LTS: TCP Timers | 325 |
| 18.1 Timers (TCP only) | 325 |
| 18.1.1 Summary | 325 |
| 18.1.2 Rules | 325 |
| – <i>timer_tt_rexmtsyn_1</i> | 325 |
| – <i>timer_tt_rexmt_1</i> | 327 |
| – <i>timer_tt_persist_1</i> | 329 |
| – <i>timer_tt_keep_1</i> | 329 |
| – <i>timer_tt_2msl_1</i> | 330 |
| – <i>timer_tt_delack_1</i> | 331 |
| – <i>timer_tt_conn_est_1</i> | 331 |
| – <i>timer_tt_fin_wait_2_1</i> | 331 |
| 19 Host LTS: UDP Input Processing | 333 |
| 19.1 Input Processing (UDP only) | 333 |
| 19.1.1 Summary | 333 |
| 19.1.2 Rules | 333 |
| – <i>deliver_in_udp_1</i> | 333 |
| – <i>deliver_in_udp_2</i> | 333 |
| – <i>deliver_in_udp_3</i> | 334 |
| 20 Host LTS: ICMP Input Processing | 335 |
| 20.1 Input Processing (ICMP only) | 335 |
| 20.1.1 Summary | 335 |
| 20.1.2 Rules | 335 |
| – <i>deliver_in_icmp_1</i> | 335 |
| – <i>deliver_in_icmp_2</i> | 336 |
| – <i>deliver_in_icmp_3</i> | 337 |
| – <i>deliver_in_icmp_4</i> | 338 |
| – <i>deliver_in_icmp_5</i> | 339 |
| – <i>deliver_in_icmp_6</i> | 339 |
| – <i>deliver_in_icmp_7</i> | 340 |
| 21 Host LTS: Network Input and Output | 341 |
| 21.1 Input and Output (Network only) | 341 |
| 21.1.1 Summary | 341 |
| 21.1.2 Rules | 341 |
| – <i>deliver_in_99</i> | 341 |
| – <i>deliver_in_99a</i> | 341 |
| – <i>deliver_out_99</i> | 341 |
| – <i>deliver_loop_99</i> | 342 |
| 22 Host LTS: BSD Trace Records and Interface State Changes | 343 |
| 22.1 Trace Records and Interface State Changes (BSD only) | 343 |
| 22.1.1 Summary | 343 |
| 22.1.2 Rules | 343 |
| – <i>trace_1</i> | 343 |
| – <i>trace_2</i> | 343 |
| – <i>interface_1</i> | 344 |
| 23 Host LTS: Time Passage | 345 |
| 23.1 Time Passage auxiliaries (TCP and UDP) | 345 |
| 23.1.1 Summary | 345 |
| 23.1.2 Rules | 345 |
| – <i>Time_Pass_timedoption</i> | 345 |
| – <i>Time_Pass_tcpcb</i> | 345 |
| – <i>Time_Pass_socket</i> | 346 |
| – <i>fmap_every</i> | 347 |
| – <i>fmap_every_pred</i> | 347 |

- *Time_Pass_host* 347
- 23.2 Host transitions with time (TCP and UDP) 348
 - 23.2.1 Summary 348
 - 23.2.2 Rules 348
 - *epsilon_1* 348
 - *epsilon_2* 348
 - *m* 348

XIV TCP1_evalSupport 350

- 24 Initial state 351**
 - 24.1 Initial state (TCP and UDP) 351
 - 24.1.1 Summary 351
 - 24.1.2 Rules 351
 - *simple_ifd_eth* 351
 - *simple_ifd_lo* 351
 - *simple_rttab* 351
 - *tid_initial* 352
 - *simple_host* 352
 - *dummy_cb* 352
 - *dummy_socket* 352
 - *dummy_sockets* 353
 - *initial_host* 353

Index 354

Part I

TCP1_utils

Chapter 1

Utility functions

This file contains various utility functions and definitions, for functions, lists, and numeric types, that are used throughout the specification.

1.1 Basic utilities

Basic utilities for functions, numbers, maps, and records.

1.1.1 Summary

| | |
|---------------------------|--|
| <i>funupd</i> | update one point of a function |
| <i>funupd_list</i> | update multiple points of a function |
| <i>clip_int_to_num</i> | clip int to num |
| <i>left_shift_num</i> | left shift, written \ll |
| <i>right_shift_num</i> | right shift, written \gg |
| <i>rounddown</i> | round v down to multiple of bs , unless $v < bs$ already |
| <i>roundup</i> | round v up to next multiple of bs ; if $v = k * bs$ then no change |
| <i>real_of_int</i> | inject int into <i>real</i> |
| <i>num_floor</i> | num floor of <i>real</i> |
| <i>num_floor_and_frac</i> | num floor and fractional part of <i>real</i> |
| <i>fm_exists</i> | finite map exists, written $\exists(k, v) :: fm.P(k, v)$ |
| <i>onlywhen</i> | used for conditional record updates |

1.1.2 Rules

– **update one point of a function:**

$$f \oplus (x \mapsto y) = \lambda x'. \text{if } x' = x \text{ then } y \text{ else } f x'$$

– **update multiple points of a function:**

$$\text{funupd_list } f \text{ } xys = \text{foldl}(\lambda f(x, y). f \oplus (x \mapsto y)) f \text{ } xys$$

– **clip int to num :**

$$\text{clip_int_to_num}(i : \text{int}) = \text{if } i < 0 \text{ then } 0 \text{ else num } i$$

– **left shift, written \ll :**

$$\text{left_shift_num}(n : \text{num})(i : \text{num}) = n * 2 ** i$$

– **right shift, written \gg :**

$$\text{right_shift_num}(n : \text{num})(i : \text{num}) = n \text{ div } 2 ** i$$

– **round v down to multiple of bs , unless $v < bs$ already :**

$$\text{rounddown } bs \ v = \text{if } v < bs \ \text{then } v \ \text{else } (v \text{ div } bs) * bs$$

– **round v up to next multiple of bs ; if $v = k * bs$ then no change :**

```

roundup bs v = ((v + (bs - 1)) div bs) * bs
– inject int into real :
real_of_int(i : int) = if i < 0 then ¬(real_of_num(num ¬i))
                      else real_of_num(num i)

– num floor of real :
num_floor(x : real) = least(n : num).real_of_num(n + 1) > x

– num floor and fractional part of real :
num_floor_and_frac(x : real)
= let n = least(n : num).real_of_num(n + 1) > x
  in
  (n, x - real_of_num n)

– finite map exists, written  $\exists(k, v) :: fm.P(k, v)$  :
fm_exists fm P =  $\exists k.k \in \text{dom}(fm) \wedge P(k, fm[k])$ 

– used for conditional record updates :
(x onlywhen b) = if b then K x else I

```

1.2 List utilities

This section contains a number of basic functions for manipulating lists.

1.2.1 Summary

| | |
|------------------------|--|
| <i>SPLIT_REV_0</i> | split worker function |
| <i>SPLIT_REV</i> | split a list after n elements, returning the reversed prefix and the remainder |
| <i>SPLIT</i> | split a list after n elements, returning the prefix and the remainder |
| <i>TAKE</i> | take the first n elements of a list |
| <i>DROP</i> | drop the first n elements of a list |
| <i>TAKEWHILE_REV</i> | split a list at first element not satisfying p , returning reversed prefix and remainder |
| <i>TAKEWHILE</i> | split a list at first element not satisfying p , returning prefix and remainder |
| <i>REPLICATE</i> | make a list of n copies of x |
| <i>decr_list</i> | decrement a list of nums by a num, dropping any that count below zero |
| <i>NOTIN'</i> | not in |
| <i>MAP_OPTIONAL</i> | map with optional result |
| <i>CONCAT_OPTIONAL</i> | concatenation of option list that drops all *s |
| <i>ORDERINGS</i> | the set of all orderings of a set |
| <i>INSERT_ORDERED</i> | insert ordered |

1.2.2 Rules

```

– split worker function:
(SPLIT_REV_0 0 ls rs = (ls, rs)) ∧
(SPLIT_REV_0(SUC n)ls(r :: rs) = SPLIT_REV_0 n(r :: ls)rs) ∧
(SPLIT_REV_0(SUC n)ls[] = (ls, []))

– split a list after  $n$  elements, returning the reversed prefix and the remainder:

```

SPLIT_REV $n\ rs = \text{SPLIT_REV_0}\ n[]rs$

– **split a list after n elements, returning the prefix and the remainder:**

SPLIT $n\ rs = \text{let } (ls, rs) = \text{SPLIT_REV}\ n\ rs \text{ in } (\text{REVERSE}\ ls, rs)$

– **take the first n elements of a list:**

TAKE $n\ rs = \text{let } (ls, rs) = \text{SPLIT_REV}\ n\ rs \text{ in } \text{REVERSE}\ ls$

– **drop the first n elements of a list:**

DROP $n\ rs = \text{let } (ls, rs) = \text{SPLIT_REV}\ n\ rs \text{ in } rs$

– **split a list at first element not satisfying p , returning reversed prefix and remainder:**

TAKEWHILE_REV $p\ ls(r :: rs) = \text{TAKEWHILE_REV}\ p(\text{if } p\ r \text{ then } (r :: ls) \text{ else } ls)rs \wedge$
 TAKEWHILE_REV $p\ ls[] = ls$

– **split a list at first element not satisfying p , returning prefix and remainder:**

TAKEWHILE $p\ rs = \text{REVERSE}(\text{TAKEWHILE_REV}\ p[]rs)$

– **make a list of n copies of x :**

(REPLICATE 0 $x = []$) \wedge

(REPLICATE($SUC\ n$) $x = x :: \text{REPLICATE}\ n\ x$)

– **decrement a list of nums by a num, dropping any that count below zero:**

((decr_list : num \rightarrow num list \rightarrow num list)

$d[] = []$) \wedge

(decr_list $d(n :: ns) = (\text{if } n < d \text{ then I else } \text{CONS}(n - d))(\text{decr_list}\ d\ ns)$)

– **not in :**

($x \notin y$) = $\neg(\text{mem}\ x\ y)$

– **map with optional result:**

MAP_OPTIONAL $f(x :: xs) = \text{append}(\text{case } f\ x \text{ of}$
 $\quad * \rightarrow []$
 $\quad || \uparrow y \rightarrow [y])$
 $(\text{MAP_OPTIONAL}\ f\ xs) \wedge$

MAP_OPTIONAL $f[] = []$

– **concatentation of option list that drops all *s:**

CONCAT_OPTIONAL $xs = \text{MAP_OPTIONAL}\ \text{I}\ xs$

– **the set of all orderings of a set :**

ORDERINGS $s\ l = (\text{list_to_set}\ l = s \wedge$

$\text{length}\ l = \text{card}\ s)$

– **insert ordered:**

INSERT_ORDERED $new\ old\ bad =$

$\text{filter}(\lambda fd. fd \in new \vee fd \in bad)old$

1.3 Assertions

This definition is an alias for false, which induces the checker to emit a special message indicating an assertion failure.

1.3.1 Summary

ASSERTION_FAILURE

assertion failure (causes checker to halt)

1.3.2 Rules

– **assertion failure (causes checker to halt) :**

ASSERTION_FAILURE(*s* : string) = **F**

Part II

TCP1_errors

Chapter 2

Error codes

This file contains the datatype of all possible error codes. The names are generally the common Unix ones; in the case of Winsock, the obvious mapping is used. Not all error codes are used in the body of the specification; those that are are described in the ‘Errors’ section of each socket call.

2.1 The type of errors

The union of all (relevant) errors on the supported architectures.

2.1.1 Summary

error

2.1.2 Rules

– :
error =

E2BIG
| EACCES
| EADDRINUSE
| EADDRNOTAVAIL
| EAFNOSUPPORT
| EAGAIN
| EWOULDBLOCK (* only used if EWOULDBLOCK ≠ EAGAIN *)
| EALREADY
| EBADF
| EBADMSG
| EBUSY
| ECANCELED
| ECHILD
| ECONNABORTED
| ECONNREFUSED
| ECONNRESET
| EDEADLK
| EDESTADDRREQ
| EDOM
| EDQUOT
| EEXIST
| EFAULT
| EFBIG
| EHOSTUNREACH

| EIDRM
| EILSEQ
| EINPROGRESS
| EINTR
| EINVAL
| EIO
| EISCONN
| EISDIR
| ELOOP
| EMFILE
| EMLINK
| EMSGSIZE
| EMULTIHOP
| ENAMETOOLONG
| ENETDOWN
| ENETRESET
| ENETUNREACH
| ENFILE
| ENOBUFS
| ENODATA
| ENODEV
| ENOENT
| ENOEXEC
| ENOLCK
| ENOLINK
| ENOMEM
| ENOMSG
| ENOPROTOOPT
| ENOSPC
| ENOSR
| ENOSTR
| ENOSYS
| ENOTCONN
| ENOTDIR
| ENOTEMPTY
| ENOTSOCK
| ENOTSUP
| ENOTTY
| ENXIO
| EOPNOTSUPP
| EOVERFLOW
| EPERM
| EPIPE
| EPROTO
| EPROTONOSUPPORT
| EPROTOTYPE
| ERANGE
| EROFS
| ESPIPE
| ESRCH
| ESTALE
| ETIME
| ETIMEDOUT
| ETXTBSY
| EXDEV
| ESHUTDOWN
| EHOSTDOWN

Part III

TCP1_signals

Chapter 3

Signal names

This file contains the datatype of signal names, with all the signals known to POSIX, Linux, and BSD. The specification does not model signal behaviour in detail, however: it treats them very nondeterministically.

3.1 The type of signals

The union of the signals supported by the target architectures. Names based on POSIX.

3.1.1 Summary

signal

3.1.2 Rules

```
- :
signal = SIGABRT
      | SIGALRM
      | SIGBUS
      | SIGCHLD
      | SIGCONT
      | SIGFPE
      | SIGHUP
      | SIGILL
      | SIGINT
      | SIGKILL
      | SIGPIPE
      | SIGQUIT
      | SIGSEGV
      | SIGSTOP
      | SIGTERM
      | SIGTSTP
      | SIGTTIN
      | SIGTTOU
      | SIGUSR1
      | SIGUSR2
      | SIGPOLL(* XSI only *)
      | SIGPROF(* XSI only *)
      | SIGSYS(* XSI only *)
      | SIGTRAP(* XSI only *)
      | SIGURG
      | SIGVTALRM(* XSI only *)
```

| SIGXCPU(* XSI only *)
| SIGXFSZ(* XSI only *)

Part IV

TCP1_baseTypes

Chapter 4

Base types

This file defines basic types used throughout the specification.

4.1 Network and OS-related types (TCP and UDP)

The specification distinguishes between the types `port` and `ip`, for which we do not use the zero values, and option types `port option` and `ip option`, with values `*` (modelling the zero values) and $\uparrow p$ and $\uparrow i$, modelling the non-zero values. Zero values are used as wildcards in some places and are forbidden in others; this typing lets that be captured explicitly.

4.1.1 Summary

port
ip
ifid
netmask
fd

4.1.2 Rules

```
— :  
port = PORT of num (* really 16 bits, non-zero *)
```

Description TCP or UDP port number, non-zero.

```
— :  
ip = ip of num (* really 32 bits, non-zero *)
```

Description IPv4 address, non-zero.

```
— :  
ifid = LO | ETH of num
```

Description Interface ID: either the loopback interface, or a numbered Ethernet interface.

– :
netmask = NETMASK of num

Description Network mask, represented as the number of 1 bits (as in a CIDR /nn suffix).

– :
 fd = FD of num

Description File descriptor. On Unix-like systems this is a small nonnegative integer; on Windows it is an arbitrary handle.

4.2 File and socket flags (TCP and UDP)

This defines the types of various flags used in the sockets API: file flags, socket flags, message flags (used in send and recv calls), and socket types (used in socket calls). The socket flags are partitioned into those with boolean, natural-number and time-valued arguments.

4.2.1 Summary

filebflag
sockbflag
socknflag
socktflag
msgbflag
socktype

4.2.2 Rules

– :
 filebflag = O_NONBLOCK
 | O_ASYNC

Description Boolean flags affecting the behaviour of an open file (or socket).
 O_NONBLOCK makes all operations on this file (or socket) nonblocking.
 O_ASYNC specifies whether signal driven I/O is enabled.

– :
 sockbflag = SO_BSDCOMPAT(* Linux only *)
 | SO_REUSEADDR
 | SO_KEEPAVIVE

```
| SO_OOBINLINE(* ? *)
| SO_DONTROUTE
```

Description Boolean flags affecting the behaviour of a socket.

SO_BSDCOMPAT Specifies whether the BSD semantics for delivery of ICMPs to UDP sockets with no peer address set is enabled.

SO_DONTROUTE Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address.

SO_KEEPAALIVE Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol.

SO_OOBINLINE Leaves received out-of-band data (data marked urgent) inline.

SO_REUSEADDR Specifies that the rules used in validating addresses supplied to `bind()` should allow reuse of local ports, if this is supported by the protocol.

Variations

| | |
|-------|--------------------------------------|
| Linux | The flag SO_BSDCOMPAT is Linux-only. |
|-------|--------------------------------------|

```
- :
socknflag = SO_SNDBUF
| SO_RCVBUF
| SO_SNDLOWAT
| SO_RCVLOWAT
```

Description Natural-number flags affecting the behaviour of a socket.

SO_SNDBUF Specifies the send buffer size.

SO_RCVBUF Specifies the receive buffer size.

SO_SNDLOWAT Specifies the minimum number of bytes to process for socket output operations.

SO_RCVLOWAT Specifies the minimum number of bytes to process for socket input operations.

```
- :
socktflag = SO_LINGER
| SO_SNDTIMEO
| SO_RCVTIMEO
```

Description Time-valued flags affecting the behaviour of a socket.

SO_LINGER specifies a maximum duration that a `close(fd)` call is permitted to block.

SO_RCVTIMEO specifies the timeout value for input operations.

SO_SNDTIMEO specifies the timeout value for an output function blocking because flow control prevents data from being sent.

```
- :
msgbflag = MSG_PEEK(* recv only, [in] *)
| MSG_OOB(* recv and send, [in] *)
| MSG_WAITALL(* recv only, [in] *)
| MSG_DONTWAIT(* recv and send, [in] *)
```

Description Boolean flags affecting the behaviour of a `send` or `recv` call.

MSG_DONTWAIT: Do not block if there is no data available.

MSG_OOB: Return out-of-band data.

MSG_PEEK: Read data but do not remove it from the socket's receive queue.

MSG_WAITALL: Block until all n bytes of data are available.

```

- :
socketype = SOCK_STREAM
          | SOCK_DGRAM

```

Description The two different flavours of socket, as passed to the `socket` call, `SOCK_STREAM` for TCP and `SOCK_DGRAM` for UDP.

4.3 Language interaction types

The specification makes almost no assumptions on the programming language used to drive sockets calls. It supposes that calls are made by threads, with thread IDs of type `tid`, and that calls return values of the `err` types indicating success or failure. Our OCaml binding maps the latter to exceptions.

Values occurring as arguments or results of sockets calls are typed. There is a HOL type `TLang_type` of the names of these types and a HOL type `TLang` which is a disjoint union of all of their values. An inductive definition defines a typing relation between the two.

4.3.1 Summary

```

tid
err
TLang_type
TLang
tlang_typing

```

4.3.2 Rules

```

- :
tid = TID of num

```

Description Thread IDs.

```

- :
err = OK of 'a | FAIL of error

```

Description Each library call returns either success (`OK v`) or failure (`FAIL err`).

```

- :
TLang_type = TLTy_INT
| TLTy_BOOL
| TLTy_STRING
| TLTy_ONE
| TLTy_PAIR of (TLang_type#TLang_type)
| TLTy_LIST of TLang_type
| TLTy_LIFT of TLang_type
| TLTy_ERR of TLang_type
| TLTy_FD
| TLTy_IP
| TLTy_PORT
| TLTy_ERROR
| TLTy_NETMASK
| TLTy_IFID
| TLTy_FILEBFLAG
| TLTy_SOCKBFLAG
| TLTy_SOCKNFLAG
| TLTy_SOCKTFLAG
| TLTy_SOCKTYPE
| TLTy_TID
| TLTy_SIGNAL

```

Description Type names for language types that are used in the sockets API.

```

- :
TLang = TL_INT of int
| TL_BOOL of bool
| TL_STRING of string
| TL_ONE of ()
| TL_PAIR of TLang#TLang
| TL_LIST of TLang list
| TL_OPTION of TLang option
| TL_ERR of TLang err
| TL_FD of fd
| TL_IP of ip
| TL_PORT of port
| TL_ERROR of error
| TL_NETMASK of netmask
| TL_IFID of ifid
| TL_FILEBFLAG of filebflag
| TL_SOCKBFLAG of sockbflag
| TL_SOCKNFLAG of socknflag
| TL_SOCKTFLAG of socktflag
| TL_SOCKTYPE of socktype
| TL_TID of tid
| TL_SIGNAL of signal

```

Description Language values.

```

- :
(∀i.tlang_typing(TL_INT i)TLTy_INT) ∧

```

```

(∀b.tlang_typing(TL_BOOL b)TLTY_BOOL) ∧
(∀s.tlang_typing(TL_STRING s)TLTY_STRING) ∧
tlang_typing(TL_ONE ())TLTY_ONE ∧
(∀p1 p2 ty1 ty2.
  tlang_typing p1 ty1 ∧ tlang_typing p2 ty2 ⇒
  tlang_typing(TL_PAIR(p1, p2))(TLTY_PAIR(ty1, ty2))) ∧
(∀tl ty.(∀e.mem e tl ⇒ tlang_typing e ty) ⇒
  tlang_typing(TL_LIST tl)(TLTY_LIST ty)) ∧
(∀p ty.tlang_typing p ty ⇒
  tlang_typing(TL_OPTION(↑ p))(TLTY_LIFT ty)) ∧
(∀ty.tlang_typing(TL_OPTION *) (TLTY_LIFT ty)) ∧
(∀e ty.tlang_typing(TL_ERR(FAIL e))(TLTY_ERR ty)) ∧
(∀p ty.tlang_typing p ty ⇒
  tlang_typing(TL_ERR(OK p))(TLTY_ERR ty)) ∧
(∀fd.tlang_typing(TL_FD fd)TLTY_FD) ∧
(∀i.tlang_typing(TL_IP i)TLTY_IP) ∧
(∀p.tlang_typing(TL_PORT p)TLTY_PORT) ∧
(∀e.tlang_typing(TL_ERROR e)TLTY_ERROR) ∧
(∀nm.tlang_typing(TL_NETMASK nm)TLTY_NETMASK) ∧
(∀ifid.tlang_typing(TL_IFID ifid)TLTY_IFID) ∧
(∀ff.tlang_typing(TL_FILEBFLAG ff)TLTY_FILEBFLAG) ∧
(∀sf.tlang_typing(TL_SOCKBFLAG sf)TLTY_SOCKBFLAG) ∧
(∀sf.tlang_typing(TL_SOCKNFLAG sf)TLTY_SOCKNFLAG) ∧
(∀sf.tlang_typing(TL_SOCKTFLAG sf)TLTY_SOCKTFLAG) ∧
(∀st.tlang_typing(TL_SOCKETYPE st)TLTY_SOCKETYPE) ∧
(∀tid.tlang_typing(TL_TID tid)TLTY_TID) ∧
(* (! ty. tlang_typing (TL_ref (Loc (ty,l))) (TLty_ref ty)) /\ *)
(* (!ex. tlang_typing (TL_exn ex) TLty_exn ) /\ *)
(* (!p ty. tlang_typing p ty ==> *)
(* tlang_typing (TL_except (EOK p)) (TLty_except ty)) /\ *)
(* (!ex ty. tlang_typing (TL_exn ex) TLty_exn ==> *)
(* tlang_typing (TL_except (EEX ex)) (TLty_except ty)) /\ *)
(∀s.tlang_typing(TL_SIGNAL s)TLTY_SIGNAL)

```

4.4 Time types

Time and duration are defined as type synonyms. Time must be non-negative and may be infinite; duration must be positive and finite.

4.4.1 Summary

| | |
|-----------------------------|-----------|
| <i>time</i> | |
| <i>type_abbrev_duration</i> | |
| <i>time_lt</i> | written < |
| <i>time_lte</i> | written ≤ |
| <i>time_gt</i> | written > |
| <i>time_gte</i> | written ≥ |

| | |
|------------------------|--------------------------|
| <i>time_min</i> | written min $x y$ |
| <i>time_max</i> | written max $x y$ |
| <i>time_plus_dur</i> | written + |
| <i>time_minus_dur</i> | written - |
| <i>real_mult_time</i> | written * |
| <i>time_zero</i> | |
| <i>duration</i> | |
| <i>abstime</i> | |
| <i>realopt_of_time</i> | |
| <i>the_time</i> | written the |

4.4.2 Rules

– :
time = ∞ | time of *real*

– :
type_abbrev duration : *real*

– **written** < :
((time_lt : time \rightarrow time \rightarrow bool)(time x)(time y) = $x < y$)
 \wedge (time_lt ∞ ys = **F**)
 \wedge (time_lt xs ∞ = **T**)

– **written** \leq :
time_lte(time x)(time y) = $x \leq y \wedge$
time_lte t ∞ = **T** \wedge
time_lte ∞ t = ($t = \infty$)

– **written** > :
time_gt xs ys = time_lt ys xs

– **written** \geq :
time_gte xs ys = time_lte ys xs

– **written** min $x y$:
time_min(time x)(time y) = time(**min** $x y$) \wedge
time_min(time x) ∞ = time $x \wedge$
time_min ∞ (time x) = time $x \wedge$
time_min ∞ ∞ = ∞

– **written** max $x y$:
time_max(time x)(time y) = time(**max** $x y$) \wedge
time_max ∞ (time x) = $\infty \wedge$
time_max(time x) ∞ = $\infty \wedge$
time_max ∞ ∞ = ∞

– **written** + :
((time_plus_dur : time \rightarrow duration \rightarrow time)
(time x) y = time($x + y$)) \wedge

(time_plus_dur ∞ $y = \infty$)

– **written** – :

((time_minus_dur : time \rightarrow duration \rightarrow time)
 (time x) $y = \text{time}(x - y)$) \wedge

(time_minus_dur ∞ $y = \infty$)

– **written** * :

(real_mult_time : real \rightarrow time \rightarrow time)

$x(\text{time } y) = \text{time}(x * y)$ \wedge

real_mult_time $x \infty = \infty$

– :

(0 : time) = time 0

– :

(duration : num \rightarrow num \rightarrow duration) $sec\ usec = \&sec + \&usec/1000000$

Description Some durations may be represented as duration $sec\ usec$, where sec and $usec$ are both natural numbers.

– :

(abstime : num \rightarrow num \rightarrow duration) $sec\ usec = \&sec + \&usec/1000000$

Description Some times may be represented as duration $sec\ usec$, where sec and $usec$ are both natural numbers.

– :

(realopt_of_time : time \rightarrow real option)(time x) = $\uparrow x \wedge$
 realopt_of_time $\infty = *$

– **written** the :

the_time(time x) = x

4.5 Basic network types: sequence numbers (TCP only)

We have several flavours of TCP sequence numbers, all represented by 32-bit values: local sequence numbers, foreign sequence numbers, and timestamps. This helps prevent confusion. We also define *tcp_seq_flip_sense*, which converts a local to a foreign sequence number and vice versa.

4.5.1 Summary

```

type_abbrev_byte
seq32
seq32_plus           written +
seq32_minus         written -
seq32_plus'         written +
seq32_minus'        written -
seq32_diff           written -
seq32_lt             written <
seq32_leq            written ≤
seq32_gt             written >
seq32_geq            written ≥
seq32_fromto
seq32_coerce
seq32_min            written min x y
seq32_max            written max x y
tcpLocal
tcpForeign
type_abbrev_tcp_seq_local
type_abbrev_tcp_seq_foreign
tcp_seq_local
tcp_seq_foreign
tcp_seq_local_to_foreign
tcp_seq_foreign_to_local
tstamp
type_abbrev_ts_seq
ts_seq

```

4.5.2 Rules

```

- :
type_abbrev byte : char

```

```

- :
seq32 = SEQ32 of 'a => word32

```

Description 32-bit wraparound sequence numbers, as used in TCP, along with their special arithmetic.

```

- written + :
seq32_plus(SEQ32 a n)(m : num) = SEQ32 a(n + n2w m)
- written - :
seq32_minus(SEQ32 a n)(m : num) = SEQ32 a(n - n2w m)
- written + :
seq32_plus'(SEQ32 a n)(m : int) = SEQ32 a(n + i2w m)
- written - :
seq32_minus'(SEQ32 a n)(m : int) = SEQ32 a(n - i2w m)
- written - :

```

seq32_diff(SEQ32(a : 'a)n)(SEQ32(b : 'a)m) = w2i(n - m)

– **written** < :

seq32_lt(n : 'a seq32)(m : 'a seq32) = ((n - m) : int) < 0

– **written** ≤ :

seq32_leq(n : 'a seq32)(m : 'a seq32) = ((n - m) : int) ≤ 0

– **written** > :

seq32_gt(n : 'a seq32)(m : 'a seq32) = ((n - m) : int) > 0

– **written** ≥ :

seq32_geq(n : 'a seq32)(m : 'a seq32) = ((n - m) : int) ≥ 0

– :

seq32_fromto(a : 'a)b(SEQ32(c : 'a)n) = SEQ32 b n

– :

seq32_coerce(SEQ32 a n) = SEQ32 ARB n

– **written min** x y :

seq32_min(n : 'a seq32)(m : 'a seq32) = **if** n < m **then** n **else** m

– **written max** x y :

seq32_max(n : 'a seq32)(m : 'a seq32) = **if** n < m **then** m **else** n

– :

tcpLocal = TCPLOCAL

– :

tcpForeign = TCPFOREIGN

– :

type_abbrev tcp_seq_local : tcpLocal seq32

– :

type_abbrev tcp_seq_foreign : tcpForeign seq32

– :

tcp_seq_local(n : word32) = SEQ32 TCPLOCAL n

– :

tcp_seq_foreign(n : word32) = SEQ32 TCPFOREIGN n

– :

tcp_seq_local_to_foreign = seq32_coerce : tcp_seq_local → tcp_seq_foreign

– :

tcp_seq_foreign_to_local = seq32_coerce : tcp_seq_foreign → tcp_seq_local

– :
tstamp = TSTAMP

|_____|

|_____|

– :
type_abbrev ts_seq : tstamp seq₃₂

|_____|

|_____|

– :
ts_seq(*n* : word₃₂) = SEQ32 TSTAMP *n*

|_____|

Part V

TCP1_netTypes

Chapter 5

Network datagram types

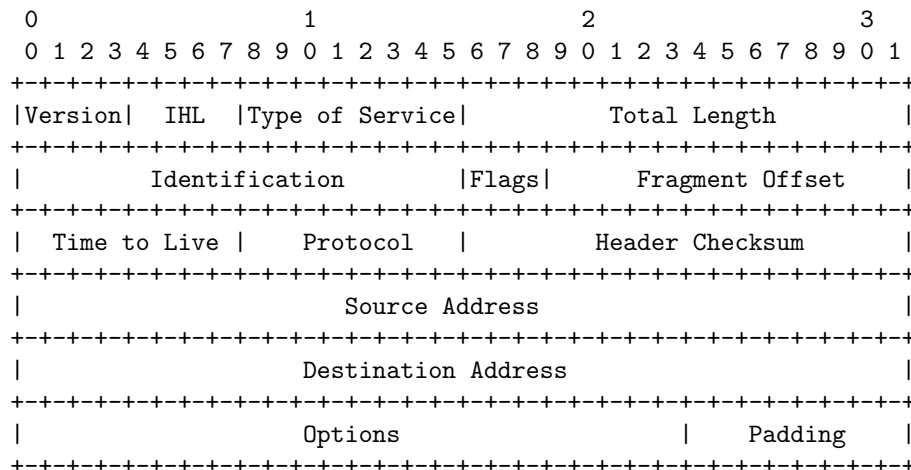
This file defines the types of the datagrams that appear on the network, with an IP message being either a TCP segment, a UDP datagram, or an ICMP datagram.

These types abstract from most fields of the IP header: version, header length, type of service, identification, DF, MF, and fragment offset, time to live, header checksum, and IP options. They faithfully model the IP header fields: protocol (TCP, UDP, or ICMP), total length, source address, and destination address. The `tcpSegment` type abstracts from the TCP checksum, reserved, and padding fields of the TCP header, from the ordering of TCP options, and from ill-formed TCP options. It faithfully models all other fields. The `udpDatagram` type abstracts from the UDP checksum but faithfully models all other fields. Lengths are represented by allowing simple lists of data bytes rather than explicit length fields. All these types collapse the encapsulation of TCP/UDP/ICMP within IP, flattening them into single records, to reduce syntactic noise throughout the specification.

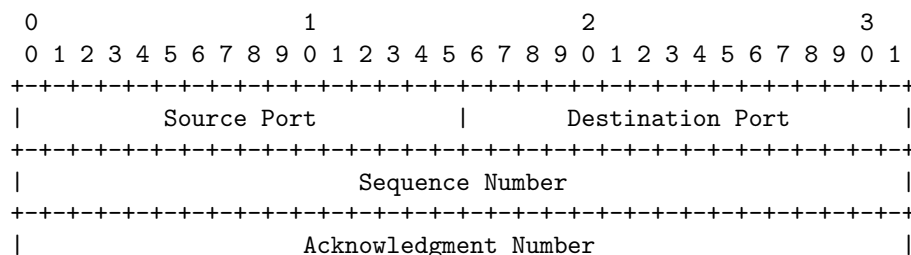
For ease of comparison we reproduce the RFC 791/793/768 header formats below.

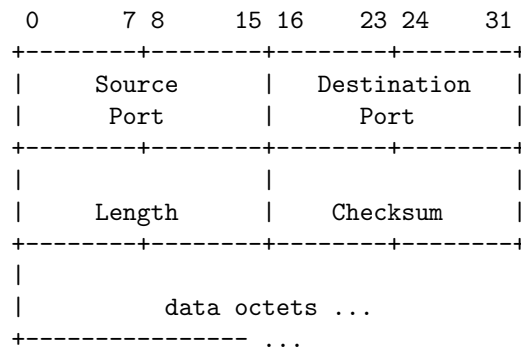
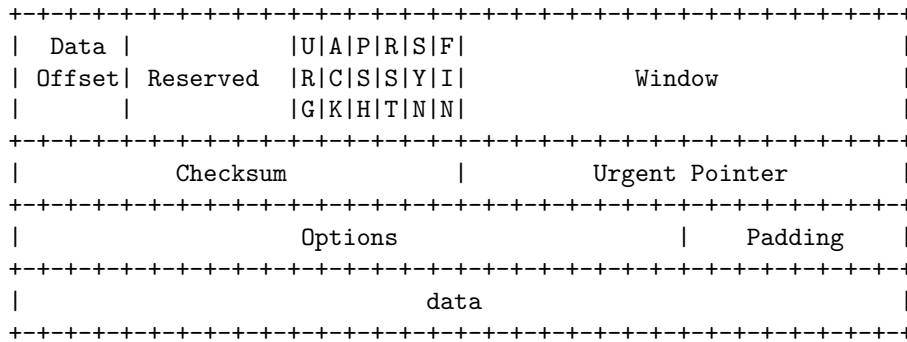
3.1. Internet Header Format

A summary of the contents of the internet header follows:



TCP Header Format





5.1 TCP segments (TCP only)

TCP segments (really *datagrams*, since we include the IP data) are modelled as follows.

5.1.1 Summary

tcpSegment
sane_seg

TCP datagram type
segment well-formedness test (physical constraints imposed by format)

5.1.2 Rules

– TCP datagram type :

tcpSegment

```

=([ is1 : ip option; (* source IP *)
  is2 : ip option; (* destination IP *)
  ps1 : port option; (* source port *)
  ps2 : port option; (* destination port *)
  seq : tcp_seq_local; (* sequence number *)
  ack : tcp_seq_foreign; (* acknowledgment number *)
  URG : bool;
  ACK : bool;
  PSH : bool;
  RST : bool;
  SYN : bool;
  FIN : bool;
  win : word16; (* window size (unsigned) *)
  ws : byte option; (* TCP option: window scaling; typically 0..14 *)

```

```

    urp : word16; (* urgent pointer (unsigned) *)
    mss : word16 option; (* TCP option: maximum segment size (unsigned) *)
    ts : (ts_seq # ts_seq) option; (* TCP option: RFC1323 timestamp value and echo-reply *)
    data : byte list
  }

```

Description The use of "local" and "foreign" here is with respect to the *sending* TCP.

– **segment well-formedness test (physical constraints imposed by format) :**
 sane_seg seg = length seg.data < (65536 – 40)

5.2 UDP datagrams (UDP only)

UDP datagrams are very simple. They are modelled as follows.

5.2.1 Summary

| | |
|--------------------|---|
| <i>udpDatagram</i> | UDP datagram type |
| <i>sane_udpdm</i> | message well-formedness test (physical constraints imposed by format) |

5.2.2 Rules

– **UDP datagram type :**
 udpDatagram
 = ((*is*₁ : ip option; (* source IP *)
 *is*₂ : ip option; (* destination IP *)
 *ps*₁ : port option; (* source port *)
 *ps*₂ : port option; (* destination port *)
 data : byte list
))

– **message well-formedness test (physical constraints imposed by format) :**
 sane_udpdm dgm = length dgm.data < (65536 – 20 – 8)

5.3 ICMP datagrams (TCP and UDP)

ICMP messages have *type* and *code* fields, both 8 bits wide. The specification deals only with some of these types, as characterised in the HOL type `icmpType` below. For each type we identify some or all of the codes that have conventional symbolic representations, but to ensure the model can faithfully represent arbitrary codes each code (HOL type) also has an `OTHER` constructor carrying a byte. The values carried are assumed not to overlap with the symbolically-represented values.

In retrospect, there seems to be no reason not to have types and codes simply particular byte constants.

5.3.1 Summary

| | |
|--------------------------------|--|
| <i>protocol</i> | protocol type for use in ICMP messages |
| <i>icmp_unreach_code</i> | |
| <i>icmp_source_quench_code</i> | |
| <i>icmp_redirect_code</i> | |
| <i>icmp_time_exceeded_code</i> | |
| <i>icmp_paramprob_code</i> | |
| <i>icmpType</i> | |
| <i>icmpDatagram</i> | ICMP datagram type |

5.3.2 Rules

```

- protocol type for use in ICMP messages :
protocol = PROTO_TCP | PROTO_UDP
    
```

```

- :
icmp_unreach_code =
NET
| HOST
| PROTOCOL
| PORT
| SRCFAIL
| NEEDFRAG of word16 option
| NET_UNKNOWN
| HOST_UNKNOWN
| ISOLATED
| NET_PROHIB
| HOST_PROHIB
| TOSNET
| TOSHOST
| FILTER_PROHIB
| PREC_VIOLATION
| PREC_CUTOFF
| OTHER of byte#word32(* really want this not to overlap *)
    
```

```

- :
icmp_source_quench_code =
QUENCH
| SQ_OTHER of byte#word32 (* writen OTHER *)
    
```

```

- :
icmp_redirect_code =
RD_NET (* written NET *)
| RD_HOST (* written HOST *)
| RD_TOSNET (* written TOSNET *)
| RD_TOSHOST (* written TOSHOST *)
| RD_OTHER of byte#word32 (* written OTHER *)
    
```

```

- :
icmp_time_exceeded_code =
INTRANS
| REASS
| TX_OTHER of byte#word32 (* written OTHER *)

- :
icmp_paramprob_code =
BADHDR
| NEEDOPT
| PP_OTHER of byte#word32 (* written OTHER *)

- :
icmpType =
ICMP_UNREACH of icmp_unreach_code
| ICMP_SOURCE_QUENCH of icmp_source_quench_code
| ICMP_REDIRECT of icmp_redirect_code
| ICMP_TIME_EXCEEDED of icmp_time_exceeded_code
| ICMP_PARAMPROB of icmp_paramprob_code
(* FreeBSD 4.6-RELEASE also does: ICMP_ECHO, ICMP_TSTMP, ICMP_MASKREQ *)

- ICMP datagram type :
icmpDatagram
= ( ( is1 : ip option; (* this is the sender of this ICMP *)
    is2 : ip option; (* this is the intended receiver of this ICMP *)

    (* we assume the enclosed IP always has at least 8 bytes of data, i.e., enough for all the fields below *)
    is3 : ip option; (* source of enclosed IP datagram *)
    is4 : ip option; (* destination of enclosed IP datagram *)
    ps3 : port option; (* source port *)
    ps4 : port option; (* destination port *)
    proto : protocol; (* protocol *)
    seq : tcp_seq_local option; (* seq *)
    t : icmpType
  )

```

5.4 IP messages (TCP and UDP)

An IP datagram is (for our purposes) either a TCP segment, an ICMP datagram, or a UDP datagram. We use the type `msg` for IP datagrams. IP datagrams may be checked for sanity, and may have their `is1` and `is2` fields inspected.

5.4.1 Summary

| | |
|-----------------|---|
| <i>msg</i> | IP message type |
| <i>sane_msg</i> | message well-formedness test (physical constraints imposed by format) |
| <i>msg_is1</i> | source IP of a message, written <i>x.is₁</i> |
| <i>msg_is2</i> | destination IP of a message, written <i>x.is₂</i> |

5.4.2 Rules

– **IP message type :**
 $\text{msg} = \text{TCP of tcpSegment} \mid \text{ICMP of icmpDatagram} \mid \text{UDP of udpDatagram}$

– **message well-formedness test (physical constraints imposed by format) :**
 $\text{sane_msg}(\text{TCP } seg) = \text{sane_seg } seg \wedge$
 $\text{sane_msg}(\text{ICMP } dgm) = \mathbf{T} \wedge$
 $\text{sane_msg}(\text{UDP } dgm') = \text{sane_udpdgm } dgm'$

– **source IP of a message, written *x.is₁* :**
 $\text{msg_is1}(\text{TCP } seg) = seg.is_1 \wedge$
 $\text{msg_is1}(\text{ICMP } dgm) = dgm.is_1 \wedge$
 $\text{msg_is1}(\text{UDP } dgm') = dgm'.is_1$

– **destination IP of a message, written *x.is₂* :**
 $\text{msg_is2}(\text{TCP } seg) = seg.is_2 \wedge$
 $\text{msg_is2}(\text{ICMP } dgm) = dgm.is_2 \wedge$
 $\text{msg_is2}(\text{UDP } dgm') = dgm'.is_2$

Part VI

TCP1_LIBinterface

Chapter 6

System call types

This file gives the system call API that is modelled by the specification.

6.1 The interface (TCP and UDP)

The Sockets API is modelled by the library interface below. As discussed in volume 1, we refine the C interface slightly:

- We use ML-style datatypes, abstracting from pointers and length parameters.
- Where the C API provides multiple entry points to a single operation (such as `send/sendto/sendmsg/write`, or `pselect/select`) we combine them all into a single general function.
- Certain special cases of general functions (such as `getsockopt` with `SO_ERROR`, `ioctl` with `SIOCATMARK`, and `fcntl` with `F_GETFL`) have been pulled out into separate functions (`getsockerr`, `socketmark` (following POSIX), and `getfileflags` respectively).
- Features not relevant to TCP or UDP (e.g. Unix domain sockets), or historical artifacts (such as the address family / protocol family distinction in `socket`) are elided.

The HOL type `LIB_interface` defines the calls. It takes their arguments to be the relevant HOL types (rather than values of `TLang`) so that HOL typechecking ensures consistency. The return types of the calls cannot be embedded so neatly within the HOL type system, so an additional `retType` function defines these (and HOL typechecking does not check this data at present).

6.1.1 Summary

LIB_interface
retType

6.1.2 Rules

```
- :  
LIB_interface =  
  accept of fd  
| bind of (fd#ip option#port option)  
| close of fd  
| connect of (fd#ip#port option)  
| disconnect of fd  
| dup of fd  
| dupfd of (fd#int)
```

```

| getfileflags of fd
| getifaddrs of ( )
| getpeername of fd
| getsockbopt of (fd#sockbflag)
| getsockerr of fd
| getsocklistening of fd
| getsockname of fd
| getsocknopt of (fd#socknflag)
| getsocktopt of (fd#socktflag)
| listen of (fd#int)
| pselect of (fd list#fd list#fd list#(int#int) option#signal list option)
| recv of (fd#int#msgbflag list)
| send of (fd#(ip#port) option#string#msgbflag list)
| setfileflags of (fd#filebflag list)
| setsockbopt of (fd#sockbflag#bool)
| setsocknopt of (fd#socknflag#int)
| setsocktopt of (fd#socktflag#(int#int) option)
| shutdown of (fd#bool#bool)
| socketmark of fd
| socket of socktype

```

Description Sockets calls with their argument types.

```

- :
retType(accept _) = TLTY_PAIR(TLTY_FD, TLTY_PAIR(TLTY_IP, TLTY_PORT))
^ retType(bind _) = TLTY_ONE
^ retType(close _) = TLTY_ONE
^ retType(connect _) = TLTY_ONE
^ retType(disconnect _) = TLTY_ONE
^ retType(dup _) = TLTY_FD
^ retType(dupfd _) = TLTY_FD
^ retType(getfileflags _) = TLTY_LIST TLTY_FILEBFLAG
^ retType(getifaddrs _) = TLTY_LIST
    (TLTY_PAIR(TLTY_IFID, TLTY_PAIR(TLTY_IP, TLTY_PAIR((TLTY_LIST TLTY_IP), TLTY_NETMASK))))
^ retType(getpeername _) = TLTY_PAIR(TLTY_IP, TLTY_PORT)
^ retType(getsockbopt _) = TLTY_BOOL
^ retType(getsockerr _) = TLTY_ONE
^ retType(getsocklistening _) = TLTY_BOOL
^ retType(getsockname _) = TLTY_PAIR(TLTY_LIFT TLTY_IP, TLTY_LIFT TLTY_PORT)
^ retType(getsocknopt _) = TLTY_INT
^ retType(getsocktopt _) = TLTY_LIFT(TLTY_PAIR(TLTY_INT, TLTY_INT))
^ retType(listen _) = TLTY_ONE
^ retType(pselect _) = TLTY_PAIR(TLTY_LIST TLTY_FD,
    TLTY_PAIR(TLTY_LIST TLTY_FD,
    TLTY_LIST TLTY_FD))
^ retType(recv _) = TLTY_PAIR(TLTY_STRING,
    TLTY_LIFT(TLTY_PAIR(TLTY_PAIR(TLTY_IP,
    TLTY_PORT),
    TLTY_BOOL)))
^ retType(send _) = TLTY_STRING
^ retType(setfileflags _) = TLTY_ONE
^ retType(setsockbopt _) = TLTY_ONE
^ retType(setsocknopt _) = TLTY_ONE
^ retType(setsocktopt _) = TLTY_ONE
^ retType(shutdown _) = TLTY_ONE
^ retType(socketmark _) = TLTY_BOOL

```

$\wedge \text{retType}(\text{socket } _) = \text{TLTY_FD}$

Description Return types of sockets calls.

6.2 Useful groups of calls (TCP and UDP)

For some purposes it is useful to group together all the system calls that expect a single `fd`, and those that expect a socket `fd`.

6.2.1 Summary

fd_op
fd_sockop

6.2.2 Rules

```

- :
fd_op fd opn = (
opn = accept(fd) ∨
(∃ is ps. opn = bind(fd, is, ps)) ∨
opn = close(fd) ∨
(∃ i p. opn = connect(fd, i, p)) ∨
opn = disconnect(fd) ∨
opn = dup(fd) ∨
(∃ fd'. opn = dupfd(fd, fd')) ∨
(opn = getfileflags(fd)) ∨
(∃ flags. opn = setfileflags(fd, flags)) ∨
opn = getsockname(fd) ∨
opn = getpeername(fd) ∨
(∃ sfb. opn = getsockbopt(fd, sfb)) ∨
(∃ sfn. opn = getsocknopt(fd, sfn)) ∨
(∃ sft. opn = getsocktopt(fd, sft)) ∨
(∃ sfb b. opn = setsockbopt(fd, sfb, b)) ∨
(∃ sfn n. opn = setsocknopt(fd, sfn, n)) ∨
(∃ sft t. opn = setsocktopt(fd, sft, t)) ∨
(∃ n. opn = listen(fd, n)) ∨
(∃ n opt. opn = recv(fd, n, opt)) ∨
(∃ data opt. opn = send(fd, data, opt)) ∨
(∃ r w. opn = shutdown(fd, r, w)) ∨
opn = socketmark(fd) ∨
opn = getsockerr(fd) ∨
opn = getsocklistening(fd)
)

```

Description Calls that expect a (single) `fd`.

```

- :
fd_sockop fd opn = (
opn = accept(fd) ∨

```

```

( $\exists is\ ps.opn = \text{bind}(fd, is, ps)$ )  $\vee$ 
( $\exists i\ p.opn = \text{connect}(fd, i, p)$ )  $\vee$ 
 $opn = \text{disconnect}(fd)$   $\vee$ 
 $opn = \text{getsockname}(fd)$   $\vee$ 
 $opn = \text{getpeername}(fd)$   $\vee$ 
( $\exists sfb.opn = \text{getsockbopt}(fd, sfb)$ )  $\vee$ 
( $\exists sfn.opn = \text{getsocknopt}(fd, sfn)$ )  $\vee$ 
( $\exists sft.opn = \text{getsocktopt}(fd, sft)$ )  $\vee$ 
( $\exists sfb\ b.opn = \text{setsockbopt}(fd, sfb, b)$ )  $\vee$ 
( $\exists sfn\ n.opn = \text{setsocknopt}(fd, sfn, n)$ )  $\vee$ 
( $\exists sft\ t.opn = \text{setsocktopt}(fd, sft, t)$ )  $\vee$ 
( $\exists n.opn = \text{listen}(fd, n)$ )  $\vee$ 
( $\exists n\ opt.opn = \text{recv}(fd, n, opt)$ )  $\vee$ 
( $\exists data\ opt.opn = \text{send}(fd, data, opt)$ )  $\vee$ 
( $\exists r\ w.opn = \text{shutdown}(fd, r, w)$ )  $\vee$ 
 $opn = \text{socketmark}(fd)$   $\vee$ 
 $opn = \text{getsockerr}(fd)$   $\vee$ 
 $opn = \text{getsocklistening}(fd)$ 
)

```

Description Calls that expect a (single) socket fd.

Part VII
TCP1_host0

Chapter 7

Host LTS labels and rule categories

This file defines the labels for the host labelled transition system, characterising the possible interactions between a host and its environment. It also defines various categories for the host LTS rules.

7.1 Transition labels (TCP and UDP)

Host transition labels.

7.1.1 Summary

Lhost0 Host transition labels

7.1.2 Rules

– **Host transition labels :**

Lhost0 =

(* library interface *)

| **LH_CALL** of *tid#LIB_interface* (* invocation of LIB call, written e.g. $\overline{\text{tid}} \cdot (\text{socket}(\text{socktype}))$ *)
| **LH_RETURN** of *tid#TLang* (* return result of LIB call, written $\overline{\text{tid}} \cdot v$ *)

(* message transmission and receipt *)

| **LH_SENDDATAGRAM** of *msg* (* output of message to the network, written $\overline{\text{msg}}$ *)
| **LH_RECVDATAGRAM** of *msg* (* input of message from the network, written *msg* *)
| **LH_LOOPDATAGRAM** of *msg* (* loopback output/input, written $\overleftarrow{\text{msg}}$ *)

(* connectivity changes *)

| **LH_INTERFACE** of *ifid#bool* (* set interface status to boolean *up*, written $\text{LH_INTERFACE}(ifid, up)$ *)

(* miscellaneous *)

| τ (* internal transition, written τ *)
| **LH_EPSILON** of *duration* (* time passage, written *dur* *)
| **LH_TRACE** of *tracerecord* (* TCP trace record, written $\text{LH_TRACE } tr$ *)

7.2 Rule categories (TCP and UDP)

A rule carries a number of flags: the protocol it relates to, its status (success, failure, or ‘bad’ failure), its category (fast or slow system call, network, etc.), and its urgency (whether it must fire immediately, or may be delayed).

7.2.1 Summary

rule_proto
rule_status
rule_cat
urgent
nonurgent
is_urgent

7.2.2 Rules

```

- :
rule_proto = RP_TCP
            | RP_UDP
            | RP_ALL

```

Description Rules are classified as to whether they relate to TCP, to UDP, or to both.

```

- :
rule_status = SUCCEEDED
            | FAIL
            | BADFAIL

```

Description Socket call rules marked SUCCEEDED construct an OK *v* value to be returned to the calling thread, whereas those marked FAIL or BADFAIL construct a FAIL *e* error to be returned. The BADFAIL rules are those involving (unusual) lack of resources, e.g. of ephemeral ports, file descriptors, or kernel memory. They are distinguished from the FAIL rules to make it easy to state properties of the form "if no bad failures occur, then...".

```

- :
rule_cat = FAST of rule_status
          | BLOCK
          | SLOW of bool => rule_status
          | NETWORK of bool
          | MISC of bool

```

Description Socket call rules are either FAST, immediately constructing a return value or error, BLOCK, entering a state in which the calling thread is blocked, or SLOW, completing processing for a blocked thread. FAST and SLOW rules have a *rule_status* as above. The NETWORK rules include message send and receive and the internal actions involved in the protocol. The MISC rules cover the remainder: returning values to threads, timer expiry, TCP tracing, interface status changes, and time passage. The *bool* argument to SLOW, NETWORK, and MISC rule categories indicates whether the rule is *urgent*. If an urgent rule is enabled then no time may pass.

– :

urgent = **T**

– :

nonurgent = **F**

– :

is_urgent(SLOW *b* _) = *b* ∧

is_urgent(NETWORK *b*) = *b* ∧

is_urgent(MISC *b*) = *b* ∧

is_urgent _ = **F**



Part VIII
TCP1_ruleids

Chapter 8

Rule names

This file defines the names of transition rules in the specification.

8.1 names (Rule only)

We list here the names of all rules in the host LTS.

8.1.1 Summary

rule_ids

8.1.2 Rules

```
- :
rule_ids = return_1
| socket_1 | socket_2
| accept_1 | accept_2 | accept_3 | accept_4 | accept_5 | accept_6 | accept_7
| bind_1 | bind_2 | bind_3 | bind_5 | bind_7 | bind_9
| close_1 | close_2 | close_3 | close_4 | close_5
| close_6 | close_7 | close_8 | close_10
| connect_1 | connect_2 | connect_3 | connect_4 | connect_4a | connect_5
| connect_5a | connect_5b | connect_5c | connect_5d | connect_6
| connect_7 | connect_8 | connect_9 | connect_10
| disconnect_1 | disconnect_2 | disconnect_3 | disconnect_4 | disconnect_5
| dup_1 | dup_2
| dupfd_1 | dupfd_3 | dupfd_4
| listen_1 | listen_1b | listen_1c | listen_2 | listen_3 | listen_4 | listen_5 | listen_7
| getfileflags_1
| setfileflags_1
| getifaddrs_1
| getsockbopt_1 | getsockbopt_2
| setsockbopt_1 | setsockbopt_2
| getsocknopt_1 | getsocknopt_4
| setsocknopt_1 | setsocknopt_4 | setsocknopt_2
| getsocktopt_1 | getsocktopt_4
| setsocktopt_1 | setsocktopt_4 | setsocktopt_5
| getsockerr_1 | getsockerr_2
| getsocklistening_1 | getsocklistening_2 | getsocklistening_3
| shutdown_1 | shutdown_2 | shutdown_3 | shutdown_4
| recv_1 | recv_2 | recv_3 | recv_4 | recv_5 | recv_6 | recv_7 | recv_8 | recv_8a | recv_9
| recv_11 | recv_12 | recv_13 | recv_14 | recv_15 | recv_16 | recv_17 | recv_20 | recv_21 | recv_22
```

| *recv_23* | *recv_24*
| *send_1* | *send_2* | *send_3* | *send_3a* | *send_4* | *send_5* | *send_5a*
| *send_6* | *send_7* | *send_8* | *send_9* | *send_10*
| *send_11* | *send_12* | *send_13* | *send_14* | *send_15*
| *send_16* | *send_17* | *send_18* | *send_19* | *send_21* | *send_22* | *send_23*
| *socketmark_1* | *socketmark_2*
| *pselect_1* | *pselect_2* | *pselect_3* | *pselect_4* | *pselect_5*
| *pselect_6*
| *getsockname_1* | *getsockname_2* | *getsockname_3*
| *getpeername_1* | *getpeername_2*
| *badf_1*
| *notsock_1*
| *intr_1*
| *resourcefail_1* | *resourcefail_2*
| *deliver_in_1* | *deliver_in_1b* | *deliver_in_2* | *deliver_in_2a*
| *deliver_in_3* | *deliver_in_3a* | *deliver_in_3b* | *deliver_in_3c*
| *deliver_in_4* | *deliver_in_5* | *deliver_in_6*
| *deliver_in_7* | *deliver_in_7a* | *deliver_in_7b* | *deliver_in_7c*
| *deliver_in_7d* | *deliver_in_8* | *deliver_in_9*
| *deliver_in_icmp_1* | *deliver_in_icmp_2* | *deliver_in_icmp_3*
| *deliver_in_icmp_4* | *deliver_in_icmp_5* | *deliver_in_icmp_6*
| *deliver_in_icmp_7*
| *deliver_in_udp_1* | *deliver_in_udp_2* | *deliver_in_udp_3*
| *deliver_in_99* | *deliver_in_99a*
| *timer_tt_rexmt_1*
| *timer_tt_rexmtsyn_1*
| *timer_tt_persist_1*
| *timer_tt_2mst_1*
| *timer_tt_delack_1*
| *timer_tt_conn_est_1*
| *timer_tt_keep_1*
| *timer_tt_fin_wait_2_1*
| *deliver_out_1*
| *deliver_out_99*
| *deliver_loop_99*
| *trace_1* | *trace_2*
| *interface_1*
| *epsilon_1*
| *epsilon_2*

Part IX

TCP1_timers

Chapter 9

Timers

This file defines the various kinds of timer that are used by the host specification. Timers are host-state components that are updated by the passage of time, in *dur* transitions. We define four kinds of timer:

1. the deadline timer (*'a timed*), which wraps a value in a timer that will count towards a (possibly fuzzy) deadline, and stop the progress of time when it reaches the maximum deadline.
2. the time-window timer (*'a timewindow*), which wraps a value in a timer just like a deadline timer, except that the value merely vanishes when it expires, rather than impeding the progress of time.
These are an optimisation, designed to avoid having an extra rule (and consequent τ transitions) just for processing the expiry of such values.
3. the ticker (*ticker*), which contains a *ts_seq* (integral wraparound 32-bit type) that is incremented by one for every time a certain interval passes. It also contains the real remainder, and the interval size that corresponds to a step.
4. the stopwatch (*stopwatch*), which may be reset at any time and counts upwards indefinitely from zero. Note it may be necessary to add some fuzziness to this timer.

For each timer we define a constructor and a time-passage function. The time-passage function takes a duration (positive real) and a timer, and returns either the timer, or $*$ if time is not permitted by the timer to pass that far (i.e., an urgent instant would be passed). Timers that never need to stop time do not return an option type. Timers that behave nondeterministically are defined relationally (taking the "result" as argument and returning a bool).

For all of them, we want the two properties defined by Lynch and Vaandrager in *Inf. and Comp.*, 128(1), 1996 (<http://theory.lcs.mit.edu/tds/papers/Lynch/IC96.html>) as S1 and S2 to hold.

9.1 Properties (TCP and UDP)

Axioms of time, that all timers must satisfy.

9.1.1 Summary

time_pass_additive
time_pass_trajectory
opttorel

9.1.2 Rules

– :
(*time_pass_additive* : (duration \rightarrow 'a \rightarrow 'a \rightarrow bool) \rightarrow bool)
time_pass

$$= \forall dur_1 \ dur_2 \ s_0 \ s_1 \ s_2. \\ time_pass \ dur_1 \ s_0 \ s_1 \wedge time_pass \ dur_2 \ s_1 \ s_2 \implies time_pass(dur_1 + dur_2) s_0 \ s_2$$

Description Property S1, additivity: If $s' \xrightarrow{d} s''$ and $s'' \xrightarrow{d'} s$ then $s' \xrightarrow{d+d'} s$.

– :
 (time_pass_trajectory : (duration \rightarrow 'a \rightarrow 'a \rightarrow bool) \rightarrow bool)
 time_pass

$$= \forall dur \ s_0 \ s_1. \\ time_pass \ dur \ s_0 \ s_1 \\ \implies \\ \exists w. \\ w \ 0 = s_0 \wedge \\ w \ dur = s_1 \wedge \\ \forall t \ t'. \\ 0 \leq t \wedge t \leq dur \wedge \\ 0 \leq t' \wedge t' \leq dur \wedge \\ t < t' \\ \implies \\ time_pass(t' - t)(w \ t)(w \ t')$$

Description Property S2 is defined as follows: Each time passage step $s' \xrightarrow{d} s$ has a *trajectory*, where a trajectory is defined as follows. If I is any left-closed interval of $\mathbb{R} \geq 0$ beginning with 0, then an I -trajectory is a function w from I to states(A) such that $w(t) \xrightarrow{t'-t} w(t')$ for all t, t' in I with $t < t'$.

Now define $w.fstate = w(0)$, $w.ltime$ to be the supremum of I , and if I is right-closed, $w.lstate = w(w.ltime)$. Then a trajectory for a step $s' \xrightarrow{d} s$ is a $[0, d]$ -trajectory with $w.fstate = s'$ and $w.lstate = s$.

In our case, S2 (which we call “trajectory”) may be stated as follows: For each time passage step $s' \xrightarrow{d} s$, there exists a function w from $[0, d]$ to states such that $w(0) = s'$, $w(d) = s$, and $w(t) \xrightarrow{t'-t} w(t')$ for all t, t' in $[0, d]$ with $t < t'$.

– :
 (opttorel : (duration \rightarrow 'a \rightarrow 'a option) \rightarrow (duration \rightarrow 'a \rightarrow 'a \rightarrow bool))
 tp dur x y
 = case tp dur x of
 ↑ $x' \rightarrow y = x'$
 || * $\rightarrow \mathbf{F}$

Description Impedance-matching coercion.

9.2 Basic timer timer (TCP and UDP)

The basic timer, `timer`, is a triple of the elapsed time, the minimum expiry time, and the maximum expiry time. It may expire at any time after the minimum expiry time, but time may not progress beyond the maximum expiry time.

9.2.1 Summary

| | |
|------------------------|---|
| <i>timer</i> | |
| <i>fuzzy_timer</i> | timer that goes off in the interval $[d - eps, d + fuz]$, like a BSD ticks-based timer |
| <i>sharp_timer</i> | timer that goes off at exactly d after now |
| <i>never_timer</i> | timer that never goes off |
| <i>upper_timer</i> | timer that goes off between now and d |
| <i>timer_expires</i> | true if the timer may expire now |
| <i>Time_Pass_timer</i> | state of timer after time passage |

9.2.2 Rules

```

- :
timer = TIMER of duration #time#time

```

```

- timer that goes off in the interval  $[d - eps, d + fuz]$ , like a BSD ticks-based timer :
(* fuz is some fuzziness added to mask the atomic nature of the model. *)

(fuzzy_timer : time → duration → duration → timer)
  d eps fuz = TIMER(0, d - eps, d + fuz)

```

```

- timer that goes off at exactly  $d$  after now :
sharp_timer d = fuzzy_timer d 0

- timer that never goes off :
never_timer = TIMER(0, ∞, ∞)

- timer that goes off between now and  $d$  :
upper_timer d = TIMER(0, 0, d)

```

```

- true if the timer may expire now :
(* NB: we assume below that this is monotonic; if it is once true it is always true (at least at any time that can be reached *)
(timer_expires : timer → bool)(TIMER(e, deadmin, deadmax))
= (time e ≥ deadmin)

```

```

- state of timer after time passage :
(Time_Pass_timer : duration → timer → timer option)
dur(TIMER(e, deadmin, deadmax))
= let e' = e + dur
  in
  if time e' ≤ deadmax
  then ↑(TIMER(e', deadmin, deadmax))
  else *

```


9.3 Deadline timer timed (TCP and UDP)

The deadline timer $'a$ timed is simply a value $'a$ annotated by a timer. This is a very convenient idiom.

9.3.1 Summary

timed
timed_val_of
timed_timer_of
timed_expires
Time_Pass_timed

9.3.2 Rules

```

- :
timed = TIMED of 'a#timer

```

```

- :
timed_val_of((x)_) = x

```

```

- :
timed_timer_of((x)_d) = d

```

```

- :
timed_expires((-)_d) = timer_expires d

```

```

- :
(Time_Pass_timed : duration → 'a timed → 'a timed option)
dur((x)_d)
= case Time_Pass_timer dur d of
  ↑ d' → ↑((x)_d')
  || * → *

```

9.4 Time-window timer timewindow (TCP and UDP)

The time-window timer $'a$ timewindow, rendered as $(x)_d^{\text{TIMEWINDOW}}$, is like a deadline timer $'a$ timed, except that when it expires the value merely evaporates, rather than causing time to stop. Thus an $'a$ timewindow never induces urgency.

9.4.1 Summary

timewindow
timewindow_val_of
timewindow_open
Time_Pass_timewindow

9.4.2 Rules

$- :$
 $\text{timewindow} = \text{TIMEWINDOW of } 'a\#\text{timer} \mid \text{TIMEWINDOWCLOSED}$

$- :$
 $\text{timewindow_val_of}((x)_{_}^{\text{TIMEWINDOW}}) = \uparrow x \wedge$
 $\text{timewindow_val_of TIMEWINDOWCLOSED} = *$

$- :$
 $\text{timewindow_open}((_)_{_}^{\text{TIMEWINDOW}}) = \mathbf{T} \wedge$
 $\text{timewindow_open TIMEWINDOWCLOSED} = \mathbf{F}$

$- :$
 $(\text{Time_Pass_timewindow} : \text{duration} \rightarrow 'a \text{ timewindow} \rightarrow 'a \text{ timewindow} \rightarrow \text{bool})$
 $\text{dur}((x)_{d}^{\text{TIMEWINDOW}})tw'$
 $= (\text{case Time_Pass_timer } dur \ d \ \text{of}$
 $\quad * \rightarrow tw' = \text{TIMEWINDOWCLOSED}$
 $\quad \uparrow d' \rightarrow tw' = (x)_{d'}^{\text{TIMEWINDOW}} \vee$
 $\quad (\text{timer_expires } d' \wedge tw' = \text{TIMEWINDOWCLOSED})) \wedge$
 $\text{Time_Pass_timewindow } dur \ \text{TIMEWINDOWCLOSED } tw' = (tw' = \text{TIMEWINDOWCLOSED})$

9.5 Ticker ticker (TCP and UDP)

A ticker ticker models a discrete time counter. It contains a counter, a remainder, a minimum duration, and a maximum duration. The counter is incremented at least once every maximum duration, and at most once every minimum duration. The remainder stores the time since the last increment.

9.5.1 Summary

ticker
ticks_of
Time_Pass_ticker
ticker_ok
tick_imin
tick_imax

9.5.2 Rules

```

- :
ticker = TICKER of ts_seq # duration (* may be zero *) # duration # duration

```

```

- :
ticks_of(TICKER(ticks, -, -, -)) = ticks

```

```

- :
(Time_Pass_ticker : duration → ticker → ticker → bool)
dur(TICKER(ticks, remdr, intvmin, intvmax))t'
= let d = remdr + dur
in
  ∃delta remdr'.
    d - real_of_num delta * intvmax ≤ remdr' ∧
    remdr' ≤ d - real_of_num delta * intvmin ∧
    0 ≤ remdr' ∧ remdr' < intvmax ∧
    t' = TICKER(ticks + delta, remdr', intvmin, intvmax)

```

```

- :
ticker_ok(TICKER(ticks, remdr, imin, imax)) =
(0 ≤ remdr ∧ remdr < imax ∧ imin ≤ imax ∧ 0 < imin)

```

```

- :
tick_imin(TICKER(t, r, imin, imax)) = imin

```

```

- :
tick_imax(TICKER(t, r, imin, imax)) = imax

```

9.6 Stopwatch stopwatch (TCP and UDP)

The stopwatch `stopwatch` records the time since it was started, with fuzziness introduced by means of a minimum and maximum rate factor applied to the passage of time.

9.6.1 Summary

stopwatch
stopwatch_val_of
Time_Pass_stopwatch

9.6.2 Rules

— :
 stopwatch = STOPWATCH of duration (* may be zero *)#real#real

— :
 stopwatch_valOf(STOPWATCH(d , -, -)) = d

— :
 (Time_Pass_stopwatch : duration \rightarrow stopwatch \rightarrow stopwatch \rightarrow bool)
 $dur(\text{STOPWATCH}(d, ratemin, ratemax))s'$
 $= \exists rate. ratemin \leq rate \wedge rate \leq ratemax \wedge$
 $s' = \text{STOPWATCH}(d + (dur * rate), ratemin, ratemax)$

Part X

TCP1_hostTypes

Chapter 10

Host types

This file defines types for the internal state of the host and its components: files, TCP control blocks, sockets, interfaces, routing table, thread states, and so on, culminating in the definition of the `host` type. It also defines TCP trace records, building on the definition of TCP control blocks.

Broadly following the implementations, each protocol endpoint has a `socket` structure which has some common fields (e.g. the associated IP addresses and ports), and some protocol-specific information.

For TCP, which involves a great deal of local state, the protocol-specific information (of type `tcp_socket`) consists of a *TCP state* (CLOSED, LISTEN, etc.), send and receive queues, and a *TCP control block*, of type `tcpcb`, with many window parameters, timers, etc. Roughly, the `socket` structure and `tcp_socket` substructure contain all the information required by most sockets rules, whereas the `tcpcb` contains fields required only by the protocol information.

10.1 Files (TCP and UDP)

10.1.1 Summary

| | |
|------------------|---|
| <i>fid</i> | file ID |
| <i>sid</i> | socket ID |
| <i>filetype</i> | type of file, with pointer to details structure |
| <i>fileflags</i> | flags set on a file |
| <i>file</i> | open file description |
| <i>File</i> | helper constructor |

10.1.2 Rules

-
- **file ID :**
fid = FID of num
 - **socket ID :**
sid = SID of num
-

Description File IDs *fid* and socket IDs *sid* are really unique, unlike file descriptors *fd*.

- **type of file, with pointer to details structure :**
filetype = FT_CONSOLE | FT_SOCKET of *sid*
- **flags set on a file :**
fileflags = { *b* : filebflag → bool }
- **open file description :**

file = { *ft* : filetype; *ff* : fileflags }

– **helper constructor :**

FILE(*ft*, *ff*) = { *ft* := *ft*; *ff* := *ff* }

Description A file is represented by an "open file description" (in POSIX terminology). This contains file flags and a file type; the specification only covers FT_CONSOLE and FT_SOCKET files. For most file types, it also contains a pointer to another structure containing data specific to that file type – in our case, a *sid* pointing to a socket structure for files of type FT_SOCKET. The file flags are defined in TCP1_baseTypes: see filebflag (p14).

10.2 TCP states (TCP only)

10.2.1 Summary

tcpstate

TCP protocol states

10.2.2 Rules

– **TCP protocol states :**

tcpstate = CLOSED

| LISTEN

| SYN_SENT

| SYN_RECEIVED

| ESTABLISHED

| CLOSE_WAIT

| FIN_WAIT_1

| CLOSING

| LAST_ACK

| FIN_WAIT_2

| TIME_WAIT

Description The states laid down by RFC793, with spelling as in the BSD source.

10.3 The TCP control block (TCP only)

10.3.1 Summary

tcpReassSegment

rextmode

rttinf

tcpcb

segment reassembly queue elements

retransmission mode

round-trip time calculation parameters

the TCP control block

10.3.2 Rules

– **segment reassembly queue elements :**

tcpReassSegment

```
=⟦ seq : tcp_seq_foreign;
   spliced_urp : tcp_seq_foreign option;
   FIN : bool;
   data : byte list
  ⟩
```

Description The TCP reassembly queue (the *t_segq* component of the TCP control block) holds information about TCP segments received out of order, pending their reassembly. It is a list of these `tcpReassSegments`, recording just the information we need about each. If a byte of urgent data has been spliced from *data* for out-of-line delivery, its sequence number is recorded in the *spliced_urp* component here to permit correct reassembly.

– **retransmission mode :**

```
rexmtmode =
  REXMTSYN
  | REXMT
  | PERSIST
```

Description TCP has three output modes: idle, retransmitting, and persisting. We introduce one more, retransmitting-syn, since the behaviour is slightly different. These modes all share the same timer, and use this "mode" parameter to distinguish. The idle mode is represented by the timer not running.

– **round-trip time calculation parameters :**

rttinf

```
=⟦ t_rttupdated : num; (* number of times rtt sampled *)
   tf_srtt_valid : bool; (* estimate is currently believed to be valid *)
   t_srtt : duration; (* smoothed round-trip time *)
   t_rttvar : duration; (* variance in round-trip time *)
   t_rttmin : duration; (* minimum rtt allowed *)
   t_lastrtt : duration; (* most recent instantaneous RTT obtained *)
   (* Note this should really be an option type which is set to * if no value has
      been obtained. The same applies to t_lastshift below. *)
   (* in BSD, this is the local variable rtt in tcp_xmit_timer(); we put it here because we don't want to store rxtcur
      in the tcpcb *)
   t_lastshift : num; (* the last retransmission shift used *)
   t_wassyn : bool (* whether that shift was REXMTSYN or not *)
   (* these two also are to avoid storing rxtcur in the tcpcb; they are somewhat annoying because they are *only*
      required for the tcp_output test that returns to slow start if the connection has been idle for >=1RTO *)
  ⟩
```

Description

This collects data used for round-trip time estimation.

tf_srtt_valid is not in BSD; instead, BSD uses *t_srtt* = 0 to indicate *t_srtt* invalid, and does horrible hacks in retransmission calculations to allow the continued use of the old *t_srtt* even after marking it invalid. We do it better!

Unlike BSD, we don't store the current retransmission interval explicitly; instead we recalculate it if it is needed.

– the TCP control block :

tcpcb = {

```

(* timers *)
tt_rexmt : (rexmtmode#num)timed option; (* retransmit timer, with mode and shift; * is idle *)
(* see tcp_output.c:356ff for more info. *)
(* as in BSD, the shift starts at zero, and is incremented each time the timer fires. So it is zero during the
first interval, 1 after the first retransmit, etc. *)
tt_keep : () timed option; (* keepalive timer *)
tt_2msl : () timed option; (* 2 * MSL TIME_WAIT timer *)
tt_delack : () timed option; (* delayed ACK timer *)
tt_conn_est : () timed option; (* connection-establishment timer, overlays keep in BSD *)
tt_fin_wait_2 : () timed option; (* FIN_WAIT_2 timer, overlays 2msl in BSD *)
t_idletime : stopwatch; (* time since last segment received *)

(* flags, some corresponding to BSD TF_ flags *)
tf_needfin : bool; (* send FIN (implicit state, used for app close while in SYN_RECEIVED) *)
tf_shouldacknow : bool; (* output a segment urgently – similar to TF_ACKNOW, but used less often*)
bsd_cantconnect : bool; (* connection establishment attempt has failed having sent a SYN – on BSD this
causes further connect() calls to fail *)

(* send variables *)
snd_una : tcp_seq_local; (* lowest unacknowledged sequence number *)
snd_max : tcp_seq_local; (* highest sequence number sent; used to recognise retransmits *)
snd_nxt : tcp_seq_local; (* next sequence number to send *)
snd_wl1 : tcp_seq_foreign; (* seq number of most recent window update segment *)
snd_wl2 : tcp_seq_local; (* ack number of most recent window update segment *)
iss : tcp_seq_local; (* initial send sequence number *)
snd_wnd : num; (* send window size: always between 0 and 65535*2**14 *)
snd_cwnd : num; (* congestion window *)
snd_ssthresh : num; (* threshold between exponential and linear snd_cwnd expansion (for slow start)*)

(* receive variables *)
rcv_wnd : num; (* receive window size *)
tf_rxwin0sent : bool; (* have advertised a zero window to receiver *)
rcv_nxt : tcp_seq_foreign; (* lowest sequence number not yet received *)
rcv_up : tcp_seq_foreign; (* received urgent pointer if any, else = rcv_nxt *)
irs : tcp_seq_foreign; (* initial receive sequence number *)
rcv_adv : tcp_seq_foreign; (* most recently advertised window *)
last_ack_sent : tcp_seq_foreign; (* last acknowledged sequence number *)

(* connection parameters *)
t_maxseg : num; (* maximum segment size on this connection *)
t_advmss : num option; (* the mss advertisement sent in our initial SYN *)
tf_doing_ws : bool; (* doing window scaling on this connection? (result of negotiation) *)
request_r_scale : num option; (* pending window scaling, if any (used during negotiation) *)
snd_scale : num; (* window scaling for send window (0..14), applied to received advertisements (RFC1323) *)
rcv_scale : num; (* window scaling for receive window (0..14), applied when we send advertisements
(RFC1323) *)

(* timestamping *)
tf_doing_tstamp : bool; (* are we doing timestamps on this connection? (result of negotiation) *)
tf_req_tstamp : bool; (* have/will request(ed) timestamps (used during negotiation) *)
ts_recent : ts_seq timewindow; (* most recent timestamp received; TimeWindowClosed if invalid. Timer
models the RFC1323 end-§4.2.3 24-day validity period. *)

(* round-trip time estimation *)
t_rttseg : (ts_seq # tcp_seq_local) option; (* start time and sequence number of segment being timed *)
t_rttinf : rttinf; (* round-trip time estimator values *)

```

```

(* retransmission *)
t_dupacks : num; (* number of consecutive duplicate acks received (typically 0..3ish; should this wrap at
64K/4G ack burst?) *)
t_badrxtwin : () timewindow; (* deadline for bad-retransmit recovery *)
snd_cwnd_prev : num; (* snd_cwnd prior to retransmit (used in bad-retransmit recovery) *)
snd_ssthresh_prev : num; (* snd_ssthresh prior to retransmit (used in bad-retransmit recovery) *)
snd_recover : tcp_seq_local; (* highest sequence number sent at time of receipt of partial ack (used in
RFC2581/RFC2582 fast recovery) *)

(* other *)
t_segq : tcpReassSegment list; (* segment reassembly queue *)
t_softerror : error option (* current transient error; reported only if failure becomes permanent *)
(* could cut this down to the actually-possible errors? *)

```

10.4 Sockets (TCP and UDP)

10.4.1 Summary

| | |
|----------------------|--|
| <i>iobc</i> | out-of-band data and status |
| <i>socket_listen</i> | extra info for a listening socket |
| <i>tcp_socket</i> | details of a TCP socket |
| <i>dgram_msg</i> | ordinary datagram on UDP receive queue |
| <i>dgram_error</i> | error (pseudo-)datagram on UDP receive queue |
| <i>dgram</i> | receive queue elements for a UDP socket |
| <i>udp_socket</i> | details of a UDP socket |
| <i>sockflags</i> | flags set on a socket |
| <i>protocol_info</i> | protocol-specific socket data |
| <i>socket</i> | details of a socket |
| <i>TCP_Sock0</i> | helper constructor |
| <i>TCP_Sock</i> | helper constructor |
| <i>UDP_Sock0</i> | helper constructor |
| <i>UDP_Sock</i> | helper constructor |
| <i>Sock</i> | helper constructor |
| <i>tcp_sock_of</i> | helper accessor (beware ARbitrary behaviour on non-TCP socket) |
| <i>udp_sock_of</i> | helper accessor (beware ARbitrary behaviour on non-UDP socket) |
| <i>proto_of</i> | helper accessor |
| <i>proto_eq</i> | compare protocol of two protocol info structures |

10.4.2 Rules

– out-of-band data and status :

```

iobc = NO_OOBDATA
| OOBDATA of byte
| HAD_OOBDATA

```

– extra info for a listening socket :

socket_listen

```
=⟨ q0 : sid list; (* incomplete connections queue *)
  q : sid list; (* completed connections queue *)
  qlimit : int(* backlog value as passed to listen *)
⟩
```

– details of a TCP socket :

tcp_socket

```
=⟨ st : tcpstate; (* here rather than in tcpcb for convenience as heavily used. Called t_state in BSD *)
  cb : tcpcb;
  lis : socket_listen option; (* invariant: * iff not LISTEN *)
  sndq : byte list;
  sndurp : num option;
  rcvq : byte list;
  rcvurp : num option; (* was "oobmark" *)
  iobc : iobc
⟩
```

– ordinary datagram on UDP receive queue :

dgram_msg

```
=⟨ data : byte list;
  is : ip option; (* source ip *)
  ps : port option(* source port *)
⟩
```

– error (pseudo-)datagram on UDP receive queue :

dgram_error

```
=⟨ e : error⟩
```

– receive queue elements for a UDP socket :

```
dgram = DGRAM_MSG of dgram_msg
       | DGRAM_ERROR of dgram_error
```

– details of a UDP socket :

udp_socket

```
=⟨ rcvq : dgram list⟩
```

Description UDP sockets are very simple – the protocol-specific content is merely a receive queue. The receive queue of a UDP socket, however, is not just a queue of bytes as it is for a TCP socket. Instead, it is a queue of *messages* and (in some implementations) *errors*. Each message contains a block of types and some ancilliary data.

Variations

| | |
|----------------|--|
| WinXP | On WinXP, errors are returned in order w.r.t. messages; this is modelled by placing them in the receive queue. |
| FreeBSD, Linux | On FreeBSD and Linux, only messages are placed in the receive queue, and errors are treated asynchronously. |

– **flags set on a socket :**

```
sockflags = {
  b : sockbflag → bool;
  n : socknflag → num;
  t : socktflag → time
}
```

– **protocol-specific socket data :**

```
protocol_info = TCP_PROTO of tcp_socket
                | UDP_PROTO of udp_socket
```

– **details of a socket :**

socket

```
= {
  fid : fid option; (* associated open file description if any *)
  sf : sockflags; (* socket flags *)
  is1 : ip option; (* local IP address if any *)
  ps1 : port option; (* local port if any *)
  is2 : ip option; (* remote IP address if any *)
  ps2 : port option; (* remote port if any *)
  es : error option; (* pending error if any *)
  cant_sndmore : bool; (* output stream ends at end of send queue *)
  cant_rcvmore : bool; (* input stream ends at end of receive queue *)
  pr : protocol_info (* protocol-specific information *)
}
```

– **helper constructor :**

```
TCP_Sock0(st, cb, lis, sndq, sndurp, rcvq, rcvurp, iobc)
= { st := st; cb := cb; lis := lis; sndq := sndq;
  sndurp := sndurp; rcvq := rcvq; rcvurp := rcvurp; iobc := iobc }
```

– **helper constructor :**

```
TCP_Sock v = TCP_PROTO(TCP_Sock0 v)
```

– **helper constructor :**

```
UDP_Sock0(rcvq) = { rcvq := rcvq }
```

– **helper constructor :**

```
UDP_Sock v = UDP_PROTO(UDP_Sock0 v)
```

– **helper constructor :**

```
SOCK(fid, sf, is1, ps1, is2, ps2, es, csm, crm, pr)
= { fid := fid; sf := sf; is1 := is1; ps1 := ps1; is2 := is2; ps2 := ps2;
  es := es; cant_sndmore := csm; cant_rcvmore := crm; pr := pr }
```

– **helper accessor (beware ARbitrary behaviour on non-TCP socket) :**

```
tcp_sock_of sock = case sock.pr of TCP_PROTO(tcp_sock) → tcp_sock || _ → ARB
```

– **helper accessor (beware ARbitrary behaviour on non-UDP socket) :**

```
udp_sock_of sock = case sock.pr of UDP_PROTO(udp_sock) → udp_sock || _ → ARB
```

– **helper accessor :**

```
proto_of(TCP_PROTO(_1)) = PROTO_TCP ∧
proto_of(UDP_PROTO(_3)) = PROTO_UDP
```

– **compare protocol of two protocol info structures :**

```
proto_eq pr pr' = (proto_of pr = proto_of pr')
```

Description Various convenience functions.

10.5 The host (TCP and UDP)

10.5.1 Summary

| | |
|----------------------------------|---|
| <i>arch</i> | the architectures we consider |
| <i>ifd</i> | network interface descriptor |
| <i>routing_table_entry</i> | routing table entry |
| <i>type_abbrev_routing_table</i> | |
| <i>bandlim_reason</i> | segment category, determining which band limiter to use |
| <i>type_abbrev_bandlim_state</i> | |
| <i>hostThreadState</i> | state of host wrt a thread |
| <i>host</i> | host details |

10.5.2 Rules

– **the architectures we consider :**

```
arch = LINUX_2_4_20_8
      | WINXP_PROF_SP1
      | FREEBSD_4_6_RELEASE
```

Description The behaviour of TCP/IP stacks varies between architectures. Here we list the architectures we consider.

In fact our FreeBSD build also has the TCP_DEBUG option turned on, and another edit to improve the accuracy of kernel time (for our automated testing). We believe that these do not impact the TCP semantics in any way.

– **network interface descriptor :**

```
ifd = ( { ipset : ip set; (* set of IP addresses of this interface *)
        primary : ip; (* and the primary IP address *)
        netmask : netmask; (* netmask *)
        up : bool (* status: up (and connected) or not *)
      } )
```

– **routing table entry :**

```
routing_table_entry = ( { destination_ip : ip;
                          destination_netmask : netmask;
                          ifid : ifid
                        } )
```

Description

Note that both routing table entries and interfaces have IP addresses (plural for interfaces, singular for RTEs) and netmasks; furthermore, interfaces have a primary IP. When we do routing, we ignore the IP addresses and mask of the interface; we only use the address and mask from the RTE. The only use of the interface info is to obtain the primary IP for use by connect().

However, there is one place where all the interface data is used: on input, the interface IP addresses are consulted to see if we can receive a packet.

The netmask of the interface is not used in the specification (except by getifaddrs()). Its function in the implementation relates to gateways etc., which (as we abstract from IP routing) we do not model.

Note that the model does not represent the routing *cache* here (i.e., cached routes with gateways, MSS, RTT, etc.), just the routing *table*. Cache data is treated nondeterministically.

```

- :
type_abbrev routing_table : routing_table_entry list

```

```

- segment category, determining which band limiter to use :

```

```

bandlim_reason = BANDLIM_UNLIMITED
                | BANDLIM_RST_CLOSEDPORT
                | BANDLIM_RST_OPENPORT

```

Description internal bandlimiter state; intended to be opaque

```

- :
type_abbrev bandlim_state : (tcpSegment# ts_seq #bandlim_reason)list

```

```

- state of host wrt a thread :

```

```

hostThreadState = RUN (* thread is running *)
                 | RET of TLang (* about to return given value to thread *)
                 | ACCEPT2 of sid (* blocked in accept *)
                 | CLOSE2 of sid (* blocked in close *)
                 | CONNECT2 of sid (* blocked in connect *)
                 | RECV2 of sid#num#msgbflag set (* blocked in recv *)
                 | SEND2 of sid#((ip#port) option#ip option#port option#ip option#port option) option
                   #byte list#msgbflag set (* blocked in send *)
                 | PSELECT2 of fd list#fd list#fd list (* blocked in pselect *)

```

Description Host threads are either RUNNING or executing a sockets call. The latter can either be about to return a value to the thread (state RET) or blocked; the remaining states capture the data required for the unblock processing for each slow call.

```

- host details :

```

```

host = [
  arch : arch; (* architecture *)
  privs : bool; (* whether process has root/CAP_NET_ADMIN privilege *)
  ifds : ifid ↦ ifd; (* interfaces *)
  rttab : routing_table; (* routing table *)
  ts : tid ↦ hostThreadState timed; (* host view of each thread state *)
  files : fid ↦ file; (* files *)
  socks : sid ↦ socket; (* sockets *)
  listen : sid list; (* list of listening sockets *)
  bound : sid list; (* list of sockets bound: head of list was first to be bound *)
  iq : msg list timed; (* input queue *)
  oq : msg list timed; (* output queue *)
  bndlm : bandlim_state; (* bandlimiting *)
  ticks : ticker; (* ticker *)
  fds : fd ↦ fid (* file descriptors (per-process) *)
]

```

Description The input and output queue timers model the interrupt scheduling delay; the first element (if any) must be processed by the timer expiry.

10.6 Trace records (TCP and UDP)

For BSD testing we make use of the BSD TCP_DEBUG option, which enables TCP debug trace records at various points in the code. This permits earlier resolution of nondeterminism in the trace checking process.

Debug records contain IP and TCP headers, a timestamp, and a copy of the implementation TCP control block. Three issues complicate their use: firstly, not all the relevant state appears in the trace record; secondly, the model deviates in its internal structures from the BSD implementation in several ways; and thirdly, BSD generates trace records in the middle of processing messages, whereas the model performs atomic transitions (albeit split for blocking invocations). These mean that in different circumstances we can use only some of the debug record fields. To save defining a whole new datatype, we reuse `tcpcb`. However, we define a special equality that only inspects certain fields, and leaves the others unconstrained.

Frustratingly, the `is1 ps1 is2 ps2` are not always available, since although the TCP control block is structure-copied into the trace record, the embedded Internet control block is not! However, in cases where these are not available, the `iss` should be sufficiently unique to identify the socket of interest.

10.6.1 Summary

| | |
|--------------------------------------|---|
| <code>traceflavour</code> | trace record flavours |
| <code>type_abbrev_tracerecord</code> | |
| <code>tracecb_eq</code> | compare two control blocks for "equality" modulo known issues |
| <code>tracesock_eq</code> | compare two sockets for "equality" modulo known issues |

10.6.2 Rules

– **trace record flavours :**

```
traceflavour = TA_INPUT
              | TA_OUTPUT
              | TA_USER
              | TA_RESPOND
              | TA_DROP
```

Description Different situations in which a trace may be generated.

– :

```
type_abbrev tracerecord : traceflavour
                        #sid
                        #(ip option(* is1 *)
                        #port option(* ps1 *)
                        #ip option(* is2 *)
                        #port option(* ps2 *)
                        ) option(* not always available! *)
                        #tcpstate(* st *)
                        #tcpcb(* cb subset *)
```

– compare two control blocks for "equality" modulo known issues :

tracecb_eq(*flav* : traceflavour)(*st* : tcpstate)(*es* : error option)(*cb* : tcpcb)(*cb'* : tcpcb)

```
= ((cb.snd_una = cb'.snd_una) ∧
  (if flav = TA_OUTPUT then T else cb.snd_max = cb'.snd_max) ∧
  (if flav = TA_OUTPUT ∨ (st = SYN_SENT ∧ es ≠ *)
  then T
  else cb.snd_nxt = cb'.snd_nxt) ∧ (* only bad on error *)
  (cb.snd_wl1 = cb'.snd_wl1) ∧
  (cb.snd_wl2 = cb'.snd_wl2) ∧
  (cb.iss = cb'.iss) ∧
  (cb.snd_wnd = cb'.snd_wnd) ∧
  (if flav = TA_OUTPUT then T else cb.snd_cwnd = cb'.snd_cwnd) ∧ (* only bad on error *)
  (cb.snd_ssthresh = cb'.snd_ssthresh) ∧
```

(* Don't check equality of *rcv_wnd*: we recalculate *rcv_wnd* lazily in *tcp_output* instead of after every successful *recv()* call, so our value is often out of date. *)

(* (if *st* = SYN_SENT then T else *cb.rcv_wnd* = *cb'.rcv_wnd*) ∧ *)

(* Removing this clause is an allowance for the fact that BSD chooses its window size rather late. *)

(* Note: we should check how it ensures that a window size it emits on a SYN retransmit is the same as on the initial transmit, and how it ensures it does not accidentally shrink the window on the next output segment (ACK of other end's SYN,ACK). *)

```
(cb.rcv_nxt = cb'.rcv_nxt) ∧
(cb.rcv_up = cb'.rcv_up) ∧
(cb.irs = cb'.irs) ∧
(if flav = TA_OUTPUT ∨ flav = TA_INPUT then T else cb.rcv_adv = cb'.rcv_adv) ∧
(if flav = TA_OUTPUT ∨ st = SYN_SENT ∨ st = TIME_WAIT
  (* we store our initially-sent MSS in t_maxseg, whereas BSD just recalculates it. This test decouples the model
  from BSD in order to cope with this. *)
  then T else cb.t_maxseg = cb'.t_maxseg) ∧ (* only bad on error *)
(cb.t_dupacks = cb'.t_dupacks) ∧
(cb.snd_scale = cb'.snd_scale) ∧
(cb.rcv_scale = cb'.rcv_scale) ∧
(* t_rttseq, if t_rtttime <> 0; ignore t_rtttime *) (* only bad on error *)
(if flav = TA_OUTPUT ∨ flav = TA_INPUT then T else
  option_map snd cb.t_rttseq = option_map snd cb'.t_rttseq) ∧
(timewindow_val_of cb.ts_recent = timewindow_val_of cb'.ts_recent) ∧
(if flav = TA_OUTPUT ∨ flav = TA_INPUT then T else cb.last_ack_sent = cb'.last_ack_sent))
(* also ignore, always: tt_delack; in case of error: tt_rexmt, t_softerror *)
```

– compare two sockets for "equality" modulo known issues :

tracesock_eq(*flav*, *sid*, *quad*, *st*, *cb*) *sid'* sock

= (proto_of *sock.pr* = PROTO_TCP ∧

let *tcp_sock* = tcp_sock_of *sock* in

sid = *sid'* ∧

(* If trace is TA_DROP then the *is₂*, *ps₂* values in the trace may not match those in the socket record — the segment is dropped because it is somehow invalid (and thus not safe to compare) *)

(case *quad* of

↑(*is₁*, *ps₁*, *is₂*, *ps₂*) → *is₁* = *sock.is₁* ∧

ps₁ = *sock.ps₁* ∧

(if *flav* = TA_DROP then T else *is₂* = *sock.is₂*) ∧

(if *flav* = TA_DROP then T else *ps₂* = *sock.ps₂*) ||

* → T) ∧

st = *tcp_sock.st* ∧

tracecb_eq flav st sock.es cb tcp_sock.cb)

Part XI

TCP1_params

Chapter 11

Host behavioural parameters

This file defines a large number of constants affecting the behaviour of the host. Many of these are adjustable by sysctls/registry keys on the target architectures.

11.1 Model parameters (TCP and UDP)

Booleans that select a particular model semantics.

11.1.1 Summary

INFINITE_RESOURCES
BSD_RTTVAR_BUG

11.1.2 Rules

```
– :  
INFINITE_RESOURCES = T
```

Description

INFINITE_RESOURCES forbids various resource failures, e.g. lack of kernel memory. These failures are nondeterministic in the specification (to be more precise the specification would have to model far more detail about the real system) and rare in practice, so for testing and reasoning one often wants to exclude them altogether.

```
– :  
BSD_RTTVAR_BUG = T
```

Description *BSD_RTTVAR_BUG* enables a peculiarity of BSD behaviour for retransmit timeouts. After *TCP_MAXRXTSHIFT* / 4 retransmit timeouts, *t_srtt* and *t_rttvar* are invalidated, but should still be used to compute future retransmit timeouts until better information becomes available. BSD makes a mistake in doing this, thus causing future retransmit timeouts to be wrong.

The code at `tcp_timer.c:420` adds the *srtt* value to the *rttvar*, shifted "appropriately", and sets *srtt* to zero. *srtt* == 0 is the indication (in BSD) that the *srtt* is invalid. We instead code this with a separate boolean, and are thus able to keep using both *srtt* and *rttvar*.

But comparing with `tcp_var.h:281`, where the values are used, reveals that the correction is in fact wrong.

This is not visible in the REXMTSYN case (where it would be most obvious), because in that case the *srtt* never was valid, and *rttvar* was cunningly hacked up to give the right value (in `tcp_subr.c:542` — and the `tcp_timer.c:420` code has no effect at all.

11.2 Scheduling parameters (TCP and UDP)

Parameters controlling the timing of the OS scheduler.

11.2.1 Summary

dschedmax
diqmax
doqmax

11.2.2 Rules

```

- :
dschedmax = time(1000/1000)(* make large for now, tighten when better understood *)
- :
diqmax = time(1000/1000)(* make large for now, tighten when better understood *)
- :
doqmax = time(1000/1000)(* make large for now, tighten when better understood *)

```

Description *dschedmax* is the maximum scheduling delay between a system call yielding a return value and that return value being passed to the process. *diqmax* and *doqmax* are the maximum scheduling delays between a message being placed on the queue and being processed (respectively, emitted). For now, pending investigation of tighter realistic upper bounds, they are all made conservatively large.

11.3 Timers (TCP and UDP)

Parameters controlling the rate and fuzziness of the various timers used in the model.

11.3.1 Summary

HZ
tickintvlmin
tickintvlmax
stopwatchfuzz
stopwatch_zero
SLOW_TIMER_INTVL
SLOW_TIMER_MODEL_INTVL
FAST_TIMER_INTVL
FAST_TIMER_MODEL_INTVL
KERN_TIMER_INTVL
KERN_TIMER_MODEL_INTVL

11.3.2 Rules

```

- :
HZ = 100 : real(* Note this is the FreeBSD value. *)

```

Description The nominal rate at which the timestamp (etc.) clock ticks, in hertz (ticks per second).

```

- :
tickintvlmin = 100/(105 * HZ) : real
- :
tickintvlmax = 105/(100 * HZ) : real

```

Description The actual bounds on the tick interval, in seconds-per-tick; must include 1/HZ, and be within the RFC1323 bounds of 1sec to 1msec.

```

- :
stopwatchfuzz = (5/100) : real(* +/- factor on accuracy of stopwatch timers *)
- :
stopwatch_zero = STOPWATCH(0, 1/(1 + stopwatchfuzz), 1 + stopwatchfuzz)

```

Description A stopwatch timer is initialised to `stopwatch_zero`, which gives it an initial time of 0 and a fuzz of `stopwatchfuzz`.

```

- :
SLOW_TIMER_INTVL = (1/2) : duration (* slow timer is 500msec on BSD *)
- :
SLOW_TIMER_MODEL_INTVL = (1/1000) : duration (* 1msec fuzziness to mask atomicity of model; Note that
it might be possible to reduce this fuzziness *)
- :
FAST_TIMER_INTVL = (1/5) : duration (* fast timer is 200msec on BSD *)
- :
FAST_TIMER_MODEL_INTVL = (1/1000) : duration (* 1msec fuzziness to mask atomicity of model; Note that
it might be possible to reduce this fuzziness *)
- :
KERN_TIMER_INTVL = tickintvlmax : duration (* precision of select timer *)
- :
KERN_TIMER_MODEL_INTVL = (the_time dschedmax) : duration (* Note that some fuzziness may be re-
quired here *)

```

(* Note this was previously 0usec fuzziness; it should really have some fuzziness, though `dschedmax` has a current value of 1s which is too high. Once `epsilon_2` is used properly by the checker, we should be able to reduce this fuzziness as it will enable the time transitions to be split. e.g. in `pselect` rules, we really want to change from `PSelect2()` to `Ret()` states pretty much exactly when the timer goes off, then allow a further `epsilon` transition before returning. *)

Description The slow, fast, and kernel timers are the timers used to control TCP time-related behaviour. The parameters here set their rates and fuzziness.

The slow timer is used for retransmit, persist, keepalive, connection establishment, `FIN_WAIT_2`, `2MSL`, and linger timers. The fast timer is used for delayed acks. The kernel timer is used for timestamp expiry, `select`, and bad-retransmit detection.

11.4 Ports, sockets, and files (TCP and UDP)

Parameters defining the classes of ports, and limits on numbers of file descriptors and sockets.

11.4.1 Summary

privileged_ports
ephemeral_ports
OPEN_MAX
OPEN_MAX_FD
FD_SETSIZE
SOMAXCONN

11.4.2 Rules

```

- :
privileged_ports = {PORT n | n < 1024}
- :
ephemeral_ports = {PORT n | n ≥ 1024 ∧ n ≤ 5000}

```

Description Ports below 1024 are reserved, and can be bound by privileged users only. Ports in the range 1024 through 5000 inclusive are used for autobinding, when no specific port is specified; these ports are called "ephemeral".

```

- :
OPEN_MAX = 957 : num (* typical value of kern.maxfilesperproc on one of our BSD boxen *)
- :
OPEN_MAX_FD = FD OPEN_MAX

```

Description A process may hold a maximum of OPEN_MAX file descriptors at any one time. These are numbered consecutively from zero on non-Windows architectures, and so the first forbidden file descriptor is OPEN_MAX_FD.

```

- :
(FD_SETSIZE : arch → num)LINUX_2_4_20_8 = 1024n ∧
FD_SETSIZE WINXP_PROF_SP1 = 64n ∧
FD_SETSIZE FreeBDS_4_6_RELEASE = 1024n

```

Description The sets of file descriptors used in calls to `pselect` can contain only file descriptors numbered less than FD_SETSIZE.

Variations

| | |
|-------|---|
| WinXP | FD_SETSIZE refers to the maximum number of file descriptors in a file descriptor set. |
|-------|---|

```

- :
SOMAXCONN = 128 : num

```

Description The maximum listen-queue length.

11.5 UDP parameters (UDP only)

UDP-specific parameters.

11.5.1 Summary

UDPpayloadMax

11.5.2 Rules

```

- :
(UDPpayloadMax : arch → num)
      LINUX_2_4_20_8 = 65507n ∧
UDPpayloadMax WINXP_PROF_SP1 = 65507n ∧
UDPpayloadMax FREEBSD_4_6_RELEASE = 9216n

```

Description The architecture-dependent maximum payload for a UDP datagram.

11.6 Buffers (TCP and UDP)

Parameters to the buffer size computation.

11.6.1 Summary

MCLBYTES size of an mbuf cluster
MSIZE
SB_MAX
oob_extra_sndbuf

11.6.2 Rules

```

- size of an mbuf cluster :
MCLBYTES = 2048 : num(* BSD default on i386; really, just needs to be >=1500 to fit an etherseg *)
- :
MSIZE = 256 : num(* BSD default on i386; really, size of an mbuf *)
- :
SB_MAX = 256 * 1024 : num(* BSD *)

```

```
– :
oob_extra_sndbuf = 1024 : num
```

11.7 File and socket flag defaults (TCP and UDP)

Default values of file and socket flags, applied on creation. Some of these are architecture-dependent. Note that SO_BSDCOMPAT should really be set to **T** by default on FreeBSD.

11.7.1 Summary

| | |
|----------------------------|------------------------------------|
| <i>ff_default_b</i> | file flags default |
| <i>ff_default</i> | |
| <i>sf_default_b</i> | bool socket flags default |
| <i>sf_default_n</i> | num socket flags defaults |
| <i>sf_default_t</i> | time socket flags defaults |
| <i>sf_default</i> | socket flags defaults |
| <i>sf_min_n</i> | minimum values of num socket flags |
| <i>sf_max_n</i> | maximum values of num socket flags |
| <i>sndrcv_timeo_t_max</i> | maximum value of send/rcv timeouts |
| <i>pselect_timeo_t_max</i> | maximum value of pselect timeouts |

11.7.2 Rules

```
– file flags default :
(ff_default_b : filebflag → bool)
    O_NONBLOCK = F ∧
ff_default_b O_ASYNC = F
– :
ff_default = { b := ff_default_b }
```

```
– bool socket flags default :
(sf_default_b : sockbflag → bool)
    SO_BSDCOMPAT = F ∧
sf_default_b SO_REUSEADDR = F ∧
sf_default_b SO_KEEPALIVE = F ∧
sf_default_b SO_OOBINLINE = F ∧
sf_default_b SO_DONTROUTE = F
```

```
– num socket flags defaults :
(sf_default_n : arch → socktype → socknflag → num)
    LINUX_2_4_20_8 SOCK_STREAM SO_SNDBUF = 16384 ∧ (* from tests *)
sf_default_n WINXP_PROF_SP1 SOCK_STREAM SO_SNDBUF = 8192 ∧ (* from tests *)
sf_default_n FREEBSD_4_6_RELEASE SOCK_STREAM SO_SNDBUF = 32 * 1024 ∧ (* from code*)
```

```
sf_default_n LINUX_2_4_20_8 SOCK_STREAM SO_RCVBUF = 43689 ∧ (* from tests - strange number? *)
sf_default_n WINXP_PROF_SP1 SOCK_STREAM SO_RCVBUF = 8192 ∧ (* from tests *)
```


sf_default_n FREEBSD_4_6_RELEASE SOCK_STREAM SO_RCVBUF = 57344 \wedge (* from code *)

sf_default_n LINUX_2_4_20_8 SOCK_STREAM SO_SNDBUF = 1 \wedge (* from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_STREAM SO_SNDBUF = 1 \wedge (* Note this value has not been checked in tests *)

sf_default_n FREEBSD_4_6_RELEASE SOCK_STREAM SO_SNDBUF = 2048 \wedge (* from code *)

sf_default_n LINUX_2_4_20_8 SOCK_STREAM SO_RCVLOWAT = 1 \wedge (* from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_STREAM SO_RCVLOWAT = 1 \wedge

sf_default_n FREEBSD_4_6_RELEASE SOCK_STREAM SO_RCVLOWAT = 1 \wedge (* from code *)

sf_default_n LINUX_2_4_20_8 SOCK_DGRAM SO_SNDBUF = 65535 \wedge (* from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_DGRAM SO_SNDBUF = 8192 \wedge (* from tests *)

sf_default_n FREEBSD_4_6_RELEASE SOCK_DGRAM SO_SNDBUF = 9216 \wedge (* from code *)

sf_default_n LINUX_2_4_20_8 SOCK_DGRAM SO_RCVBUF = 65535 \wedge (* correct from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_DGRAM SO_RCVBUF = 8192 \wedge (* correct from tests *)

sf_default_n FREEBSD_4_6_RELEASE SOCK_DGRAM SO_RCVBUF = 42080 \wedge (* from tests but:
41600 from code; i386
only as dependent on
sizeof(struct sock-
addr_in) *)

sf_default_n LINUX_2_4_20_8 SOCK_DGRAM SO_SNDBUF = 1 \wedge (* from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_DGRAM SO_SNDBUF = 1 \wedge (* from tests *)

sf_default_n FREEBSD_4_6_RELEASE SOCK_DGRAM SO_SNDBUF = 2048 \wedge (* from code *)

sf_default_n LINUX_2_4_20_8 SOCK_DGRAM SO_RCVLOWAT = 1 \wedge (* from tests *)

sf_default_n WINXP_PROF_SP1 SOCK_DGRAM SO_RCVLOWAT = 1 \wedge (* from tests *)

sf_default_n FREEBSD_4_6_RELEASE SOCK_DGRAM SO_RCVLOWAT = 1 \wedge (* from code *)

– **time socket flags defaults :**

(sf_default_t : socktflag \rightarrow time)

SO_LINGER = ∞ \wedge

sf_default_t SO_SNDTIMEO = ∞ \wedge

sf_default_t SO_RCVTIMEO = ∞

– **socket flags defaults :**

```
sf_default arch socktype = {
  b := sf_default_b;
  n := sf_default_n arch socktype;
  t := sf_default_t
}
```

– **minimum values of num socket flags :**

(sf_min_n : arch \rightarrow socknflag \rightarrow num)

LINUX_2_4_20_8 SO_SNDBUF = 2048 \wedge (* from tests *)

sf_min_n WINXP_PROF_SP1 SO_SNDBUF = 0 \wedge (* from tests *)

sf_min_n FREEBSD_4_6_RELEASE SO_SNDBUF = 1 \wedge (* from code *)

sf_min_n LINUX_2_4_20_8 SO_RCVBUF = 256 \wedge (* from tests *)

sf_min_n WINXP_PROF_SP1 SO_RCVBUF = 0 \wedge (* from tests *)

sf_min_n FREEBSD_4_6_RELEASE SO_RCVBUF = 1 \wedge (* from code *)

sf_min_n LINUX_2_4_20_8 SO_SNDBUF = 1 \wedge (* from tests *)

```

sf_min_n WINXP_PROF_SP1 SO_SNDLOWAT = 1 ^ (* Note this value has not been checked in testing. *)
sf_min_n FREEBSD_4_6_RELEASE SO_SNDLOWAT = 1 ^ (* from code *)
sf_min_n LINUX_2_4_20_8 SO_RCVLOWAT = 1 ^ (* from tests *)
sf_min_n WINXP_PROF_SP1 SO_RCVLOWAT = 1 ^ (* Note this value has not been checked in testing. *)
sf_min_n FREEBSD_4_6_RELEASE SO_RCVLOWAT = 1(* from code *)

```

– **maximum values of num socket flags :**

(sf_max_n : arch → socknflag → num)

```

LINUX_2_4_20_8 SO_SNDBUF = 131070 ^ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_SNDBUF = 131070 ^ (* from tests *)
sf_max_n FREEBSD_4_6_RELEASE SO_SNDBUF =
  SB_MAX * MCLBYTES div(MCLBYTES + MSIZE) ^ (* from code *)
sf_max_n LINUX_2_4_20_8 SO_RCVBUF = 131070 ^ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_RCVBUF = 131070 ^ (* from tests *)
sf_max_n FREEBSD_4_6_RELEASE SO_RCVBUF =
  SB_MAX * MCLBYTES div(MCLBYTES + MSIZE) ^ (* from code *)
sf_max_n LINUX_2_4_20_8 SO_SNDLOWAT = 1 ^ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_SNDLOWAT = 1 ^ (* Note this value has not been checked in testing. *)
sf_max_n FREEBSD_4_6_RELEASE SO_SNDLOWAT =
  SB_MAX * MCLBYTES div(MCLBYTES + MSIZE) ^ (* clip to SO_SNDBUF *)
sf_max_n LINUX_2_4_20_8 SO_RCVLOWAT = w2n INT32_SIGNED_MAX ^ (* from code *)
sf_max_n WINXP_PROF_SP1 SO_RCVLOWAT = 1 ^ (* Note this value has not been checked in testing. *)
sf_max_n FREEBSD_4_6_RELEASE SO_RCVLOWAT =
  SB_MAX * MCLBYTES div(MCLBYTES + MSIZE)(* clip to SO_RCVBUF *)

```

– **maximum value of send/recv timeouts :**

sndrcv_timeo_t_max = time 655350000

– **maximum value of pselect timeouts :**

pselect_timeo_t_max = time(31 * 24 * 3600)

11.8 RFC-specified limits (TCP only)

Protocol value limits specified in the TCP RFCs.

11.8.1 Summary

| | |
|-------------------------|--|
| <i>dtsinval</i> | RFC1323 s4.2.3: timestamp validity period. |
| <i>TCP_MAXWIN</i> | maximum (scaled) window size |
| <i>TCP_MAXWINSSCALE</i> | maximum window scaling exponent |

11.8.2 Rules

– **RFC1323 s4.2.3: timestamp validity period. :**

dtsinval = time(24 * 24 * 60 * 60)

– **maximum (scaled) window size :**

TCP_MAXWIN = 65535 : num

- **maximum window scaling exponent :**
TCP_MAXWINSIZE = 14 : num

Description The maximum (scaled) window size value is TCP_MAXWIN, and the maximum scaling exponent is TCP_MAXWINSIZE. Thus the maximum window size is TCP_MAXWIN \ll TCP_MAXWINSIZE.

11.9 Protocol parameters (TCP only)

Various TCP protocol parameters, many adjustable by `sysctl` settings (or equivalent). The values here are typical. It was not considered worthwhile modelling these parameters changing during operation.

11.9.1 Summary

| | |
|-----------------------|--|
| <i>MSSDFLT</i> | initial <i>t_maxseg</i> , modulo route and link MTUs |
| <i>SS_FLTSZ_LOCAL</i> | initial <i>snd_cwnd</i> for local connections |
| <i>SS_FLTSZ</i> | initial <i>snd_cwnd</i> for non-local connections |
| <i>TCP_DO_NEWRENO</i> | do NewReno fast recovery |
| <i>TCP_Q0MINLIMIT</i> | |
| <i>TCP_Q0MAXLIMIT</i> | |
| <i>backlog_fudge</i> | |

11.9.2 Rules

-
- **initial *t_maxseg*, modulo route and link MTUs :**
MSSDFLT = 512 : num(* BSD default; RFC1122 sec. 4.2.2.6 says this MUST be 536 *)

-
- **initial *snd_cwnd* for local connections :**
SS_FLTSZ_LOCAL = 4 : num(* BSD; is a sysctl *)
 - **initial *snd_cwnd* for non-local connections :**
SS_FLTSZ = 1 : num(* BSD; is a sysctl *)

-
- **do NewReno fast recovery :**
TCP_DO_NEWRENO = T : bool(* BSD default *)

-
- **:**
TCP_Q0MINLIMIT = 30 : num(* FreeBSD 4.6-RELEASE: tcp_synccache.bucket_limit *)
 - **:**
TCP_Q0MAXLIMIT = 512 * 30 : num(* FreeBSD 4.6-RELEASE: tcp_synccache.cache_limit *)
-

Description The incomplete-connection listen queue q_0 has a nondeterministic length limit. Connections *may* be dropped once q_0 reaches TCP_Q0MINLIMIT, and *must* be dropped once q_0 reaches TCP_Q0MAXLIMIT.

```

- :
backlog_fudge( $n$  : int) = min SOMAXCONN(clip_int_to_num  $n$ )

```

Description The backlog length fudge-factor function, which translates the requested length of the listen queue into the actual value used. Some architectures apply a linear transformation here.

11.10 Time values (TCP only)

Various time intervals controlling TCP's behaviour.

11.10.1 Summary

```

TCPTV_DELACK
TCPTV_RTOBASE
TCPTV_RTTVARBASE
TCPTV_MIN
TCPTV_REXMTMAX
TCPTV_MSL
TCPTV_PERSMIN
TCPTV_PERSMAX
TCPTV_KEEP_INIT
TCPTV_KEEP_IDLE
TCPTV_KEEPINTVL
TCPTV_KEEPCNT
TCPTV_MAXIDLE

```

11.10.2 Rules

```

- :
TCPTV_DELACK = time(1/10)(* FreeBSD 4.6-RELEASE, tcp_timer.h *)

```

```

- :
TCPTV_RTOBASE = 3 : duration (* initial RTT, in seconds: FreeBSD 4.6-RELEASE, tcp_timer.h *)

- :
TCPTV_RTTVARBASE = 0 : duration (* initial retransmit variance, in seconds *)
(* FreeBSD has no way of encoding an initial RTT variance, but we do (thanks to tf_srttvalid); it should be zero so
TCPTV_RTOBASE = initial RTO *)

- :
TCPTV_MIN = 1 : duration (* minimum RTT in absence of cached value, in seconds: FreeBSD 4.6-RELEASE, tcp_timer.h *)

- :
TCPTV_REXMTMAX = time 64(* BSD: maximum possible RTT *)

```

```

- :
TCPTV_MSL = time 30(* maximum segment lifetime: BSD: tcp_timer.h:79 *)

- :
TCPTV_PERSMIN = time 5(* BSD: minimum possible persist interval: tcp_timer.h:85 *)

- :
TCPTV_PERSMAX = time 60(* BSD: maximum possible persist interval: tcp_timer.h:86 *)

- :
TCPTV_KEEP_INIT = time 75(* connect timeout: BSD: tcp_timer.h:88 *)

- :
TCPTV_KEEP_IDLE = time(120 * 60)(* time before first keepalive probe: BSD: tcp_timer.h:89 *)

- :
TCPTV_KEEPINTVL = time 75(* time between subsequent keepalive probes: BSD: tcp_timer.h:90 *)

- :
TCPTV_KEEPCNT = 8 : num(* max number of keepalive probes (+/- a few?): BSD: tcp_timer.h:91 *)

- :
TCPTV_MAXIDLE = 8 * TCPTV_KEEPINTVL (* BSD calls this tcp_maxidle *)

```

11.11 Timing-related parameters (TCP only)

Parameters relating to TCP's exponential backoff.

11.11.1 Summary

| | |
|-------------------------------|--|
| <i>TCP_BSD_BACKOFFS</i> | TCP exponential retransmit backoff: BSD: from source code, tcp_timer.c:155 |
| <i>TCP_LINUX_BACKOFFS</i> | TCP exponential retransmit backoff: Linux: experimentally determined |
| <i>TCP_WINXP_BACKOFFS</i> | TCP exponential retransmit backoff: WinXP: experimentally determined |
| <i>TCP_MAXRXTSHIFT</i> | TCP maximum retransmit shift |
| <i>TCP_SYNACKMAXRXTSHIFT</i> | TCP maximum SYNACK retransmit shift |
| <i>TCP_SYN_BSD_BACKOFFS</i> | TCP exponential SYN retransmit backoff: BSD: tcp_timer.c:152 |
| <i>TCP_SYN_LINUX_BACKOFFS</i> | TCP exponential SYN retransmit backoff: Linux: experimentally determined |
| <i>TCP_SYN_WINXP_BACKOFFS</i> | TCP exponential SYN retransmit backoff: WinXP: experimentally determined |

11.11.2 Rules

-
- **TCP exponential retransmit backoff: BSD: from source code, tcp_timer.c:155 :**
TCP_BSD_BACKOFFS = [1; 2; 4; 8; 16; 32; 64; 64; 64; 64; 64; 64] : num list
 - **TCP exponential retransmit backoff: Linux: experimentally determined :**
TCP_LINUX_BACKOFFS = [1; 2; 4; 8; 16; 32; 64; 128; 256; 512; 512] : num list(* Note: the tail may be incomplete *)
 - **TCP exponential retransmit backoff: WinXP: experimentally determined :**
TCP_WINXP_BACKOFFS = [1; 2; 4; 8; 16] : num list(* Note: the tail may be incomplete *)
-
- **TCP maximum retransmit shift :**
TCP_MAXRXTSHIFT = 12 : num(* TCPv2p842 *)
 - **TCP maximum SYNACK retransmit shift :**
TCP_SYNACKMAXRXTSHIFT = 3 : num(* FreeBSD 4.6-RELEASE, tcp_synccache.c:SYNCCACHE_MAXREXMTS *)
-
- **TCP exponential SYN retransmit backoff: BSD: tcp_timer.c:152 :**
TCP_SYN_BSD_BACKOFFS = [1; 1; 1; 1; 1; 2; 4; 8; 16; 32; 64; 64; 64] : num list(* Our experimentation shows that this list stops at 8. This will be due to the connection establishment timer firing. Values here are obtained from the BSD source *)
 - **TCP exponential SYN retransmit backoff: Linux: experimentally determined :**
TCP_SYN_LINUX_BACKOFFS = [1; 2; 4; 8; 16] : num list(* This list might be longer. Experimentation does not show further entries, perhaps due to the connection establishment timer firing *)
 - **TCP exponential SYN retransmit backoff: WinXP: experimentally determined :**
TCP_SYN_WINXP_BACKOFFS = [1; 2] : num list(* This list might be longer. Experimentation does not show further entries, perhaps due to the connection establishment timer firing *)
-

Part XII

TCP1_auxFns

Chapter 12

Auxiliary functions

This file defines a large number of auxiliary functions to the host specification.

12.1 Architecture handling (TCP and UDP)

Many aspects of host behaviour differ from one OS to another, and so a host has an architecture parameter detailing its precise OS and version (e.g., LINUX_2.4_20.8). Very often, however, we do not need to be so precise – a certain behaviour might apply to all Linux, or even all Unix, OSes. Below we define predicates for these cases, to allow variant architectures to be easily added later.

12.1.1 Summary

| | |
|---------------------|--------------------------------------|
| <i>windows_arch</i> | test if host architecture is Windows |
| <i>bsd_arch</i> | test if host architecture is BSD |
| <i>linux_arch</i> | test if host architecture is Linux |
| <i>unix_arch</i> | test if host architecture is Unix |

12.1.2 Rules

-
- **test if host architecture is Windows :**
`windows_arch arch = (arch ∈ {WINXP_PROF_SP1})`
 - **test if host architecture is BSD :**
`bsd_arch arch = (arch ∈ {FREEBSD_4.6_RELEASE})`
 - **test if host architecture is Linux :**
`linux_arch arch = (arch ∈ {LINUX_2.4_20.8})`
 - **test if host architecture is Unix :**
`unix_arch arch = (arch ∈ {LINUX_2.4_20.8; FREEBSD_4.6_RELEASE})`
-

12.2 Interfaces and IP addresses (TCP and UDP)

Constructors, predicates, and helper functions that deal with interfaces, IP addresses, and routing.

12.2.1 Summary

| | |
|------------------|---|
| <i>mask</i> | apply a netmask to an IP to obtain the network number |
| <i>mask_bits</i> | compute network bitmask from netmask |

| | |
|----------------------------|--|
| <i>IP</i> | constructor for dotted-decimal IP addresses |
| <i>IN_MULTICAST</i> | the set of multicast addresses |
| <i>INADDR_BROADCAST</i> | the local broadcast address |
| <i>LOOPBACK_ADDRS</i> | the set of loopback addresses |
| <i>ip_localhost</i> | the canonical loopback address, aka 'localhost' |
| <i>in_loopback</i> | is IP address a loopback address? |
| <i>in_local</i> | is IP address a local address? |
| <i>local_ips</i> | the set of local IP addresses |
| <i>local_primary_ips</i> | the set of local primary IP addresses |
| <i>is_localnet</i> | is IP address on a local subnet of this host? |
| <i>if_broadcast</i> | is IP address a broadcast address? |
| <i>if_any</i> | the set of addresses in an interface's subnet |
| <i>is_broadormulticast</i> | is IP address a broadcast/multicast address? |
| <i>routeable</i> | compute set of routeable addresses for a routing table entry |
| <i>outroute_ifids</i> | determine list of possible sending interfaces |
| <i>ifid_up</i> | is the interface up? |
| <i>outroute</i> | compute interface to use to send to given IP, if any |
| <i>auto_outroute</i> | compute source address to use to route to given IP |
| <i>test_outroute_ip</i> | test if we can route to given IP, returning appropriate error if not |
| <i>test_outroute</i> | if destination IP specified, do test_outroute_ip |
| <i>loopback_on_wire</i> | check if a message bears a loopback address |

12.2.2 Rules

– **apply a netmask to an IP to obtain the network number :**
 $\text{mask}(\text{NETMASK } m)(\text{ip } n) = \text{ip}((n \text{ div}(2^{**} (32 - m))) * 2^{**} (32 - m))$

– **compute network bitmask from netmask :**
 $\text{mask_bits}(\text{NETMASK } m) = ((2^{**} 32 - 1) \text{ div}(2^{**} (32 - m))) * 2^{**} (32 - m)$

Description Netmask operations. Recall netmasks are stored as the number of 1 bits in the mask; thus 255.255.128.0 is modelled by NETMASK 17.

– **constructor for dotted-decimal IP addresses :**
 $\text{IP}(a : \text{num})(b : \text{num})(c : \text{num})(d : \text{num}) = \text{ip}(a * 2^{**} 24 + b * 2^{**} 16 + c * 2^{**} 8 + d)$

– **the set of multicast addresses :**
 $\text{IN_MULTICAST} = \{i \mid \text{mask}(\text{NETMASK } 4)i = \text{IP } 224 \ 0 \ 0 \ 0\}$

– **the local broadcast address :**
 $\text{INADDR_BROADCAST} = \text{IP } 255 \ 255 \ 255 \ 255$

– **the set of loopback addresses :**
 $\text{LOOPBACK_ADDRS} = \{i \mid \text{mask}(\text{NETMASK } 8)i = \text{IP } 127 \ 0 \ 0 \ 0\}$

– **the canonical loopback address, aka 'localhost' :**
 $\text{ip_localhost} = \text{IP } 127 \ 0 \ 0 \ 1$

– **is IP address a loopback address? :**
 $\text{in_loopback } i = (i \in \text{LOOPBACK_ADDRS})$

– **is IP address a local address? :**
 $\text{in_local}(ifds : ifid \mapsto ifd)i =$
 $(\text{in_loopback } i \vee$
 $i \in (\mathbf{bigunion}\{ifd_ipset \mid ifd_ \in (\mathbf{rng}(ifds))\}))$

(* Note: the test "in_loopback *i*" is usually redundant as there is almost always a loopback interface in *ifds* with *ipset* = LOOPBACK_ADDRS *)

– **the set of local IP addresses :**

$\text{local_lips}(ifds : ifid \mapsto ifd) = \mathbf{bigunion}\{ifd_ipset \mid ifd_ \in (\mathbf{rng}(ifds))\}$

(* annoying: ifd is a constructor, and { | } has no binder to allow us to shadow it *)

– **the set of local primary IP addresses :**

$\text{local_primary_ips}(ifds : ifid \mapsto ifd) = \{ifd_primary \mid ifd_ \in (\mathbf{rng}(ifds))\}$

– **is IP address on a local subnet of this host? :**

$\text{is_localnet}(ifds_0 : ifid \mapsto ifd)i =$

$(\exists ifd.ifd \in (\mathbf{rng}(ifds_0)) \wedge \text{mask } ifd.netmask \ i = \text{mask } ifd.netmask \ ifd.primary)$

– **is IP address a broadcast address? :**

$\text{if_broadcast}(ifd0 : ifd)$

= **case** (*ifd0.netmask*, *mask ifd0.netmask ifd0.primary*) **of**
 (NETMASK *m*, *ip n* (* *n* has been masked by *m* above *)) \rightarrow
 $\text{ip}(n + 2^{**} (32 - m) - 1)$

(* Note: would be much easier if IPs were actually *word32* rather than *num* *)

(* corresponds to INADDR_BROADCAST for the interface *)

– **the set of addresses in an interface's subnet :**

$\text{if_any}(ifd0 : ifd)$

= **case** (*ifd0.netmask*, *mask ifd0.netmask ifd0.primary*) **of**
 (NETMASK *m*, *ip n* (* *n* has been masked by *m* above *)) \rightarrow
 $\text{ip}(n)$

(* Note: would be much easier if IPs were actually *word32* rather than *num* *)

Description Various distinguished IP addresses and sets of IP addresses. Some of these are dependent on the host's set of interfaces.

– **is IP address a broadcast/multicast address? :**

$\text{is_broadmcast}(ifds_0 : ifid \mapsto ifd)i =$

$(i \in \text{IN_MULTICAST} \vee (* \text{ is } i \text{ a multicast address? } *))$

$i = \text{INADDR_BROADCAST} \vee (* \text{ is } i \text{ the default broadcast address? [CORRECT NAME?] } *)$

$\exists(k, ifd0) :: ifds_0.$

$i \in \{\text{if_broadcast } ifd0; (* \text{ is } i \text{ the broadcast addr for any interface? } *)$

$\text{if_any } ifd0\}$ (* RFC 1122 - should accept an all-0s or all-1s broadcast address. all three Oses do *)

Description Test if IP address *i* is a broadcast or multicast address, wrt the given set of interfaces *ifds₀*. If no interfaces given (*ifds₀* = *), then treat only INADDR_BROADCAST as a broadcast address.

These correctly use the interface rather than the routing-table entry to check what is a broadcast address and what is in the local net of this host. Whether there is a route allowing a send to that local net is another question entirely, although the two data structures *should* be consistent.

– **compute set of routeable addresses for a routing table entry :**

$\text{routeable}(rte : \text{routing_table_entry}) =$

$\{i \mid \text{mask } rte.destination_netmask \ i = \text{mask } rte.destination_netmask \ rte.destination_ip\}$

– **determine list of possible sending interfaces :**

$\text{outroute_ifids}(i_2, rttab : \text{routing_table}) =$

$\text{MAP_OPTIONAL}(\lambda rte.\text{if } i_2 \in \text{routeable } rte \text{ then } \uparrow \text{rte.ifid} \text{ else } *)rttab$

Description Determine the list of possible interfaces to use in sending to a given IP, based on the routing table.

```

– is the interface up? :
ifid_up ifds ifid = (ifds[ifid]).up

– compute interface to use to send to given IP, if any :
outroute(i2, rttab : routing_table, ifds : ifid ↦ ifd) =
case filter(ifid_up ifds)(outroute_ifids(i2, rttab)) of
  [] → *
  || (ifid :: _987) → ↑ ifid

```

Description Determine the interface to use to send to a given IP, if possible. Returns the first up interface that can route to the destination.

```

– compute source address to use to route to given IP :
auto_outroute(i2', ↑ i2, rttab, ifds) = {i2} ∧
auto_outroute(i2', *, rttab, ifds) = case outroute(i2', rttab, ifds) of
  ↑ ifid → {(ifds[ifid]).primary}
  || * → {}

```

Description Compute source address to use to route to a given IP, if any possible. If the caller provides an address, use that without checking; otherwise try to find one. Do not return a specific error code. Used for autobinding to a local IP address.

```

– test if we can route to given IP, returning appropriate error if not :
test_outroute_ip(i2 : ip, rttab, ifds, arch)
= let ifds = outroute_ifids(i2, rttab) in
  if ifds = [] then
    (if linux_arch arch then ↑ ENETUNREACH
     else ↑ EHOSTUNREACH)
  else
    if filter(ifid_up ifds)ifds = [] then
      ↑ ENETDOWN
    else *

– if destination IP specified, do test_outroute_ip :
test_outroute(msg : msg, rttab, ifds, arch)
= case msg.is2 of
  ↑ i2 → ↑(test_outroute_ip(i2, rttab, ifds, arch))
  || _ → *

```

Description Check that we can route the message out. First check that there is an interface that can route to the destination address. If not, EHOSTUNREACH. Then, check that there is one of these that is up. If not, ENETDOWN. Otherwise, succeed (indicated by empty set of possible errors). The message should have i_2 specified.

You might think that we should check that the interface can send from the source address also, but in fact, in the weak end system model, they don't need to be the same interface. We have tested Linux, and find this behaviour. Not sure yet about BSD, but suspect it will be the same. test 20030204T1525 or so.

test_outroute modified to be functional rather than relational, as behaviour is purely deterministic. The result is of type `error option option`, where the first level of "optionality" indicates whether or not the function is even being called on valid input (whether or not message has an `is2` "field"), and the next level indicates errors being raised, or not.

Note that if we "knew" that this would only be called on messages with ok is_2 fields, then it would be easier still to just use **the**, ignore the fact that the function had an unspecified result on arguments with bad is_2 fields, and make the result type `error option`.

– **check if a message bears a loopback address :**

```
loopback_on_wire(msg : msg)(ifds : ifid ↦ ifd) =
case (msg.is1, msg.is2) of
  (*, *) → F
  || (*, ↑ j) → F
  || (↑ i, *) → F
  || (↑ i, ↑ j) → in_loopback i ∧ ¬in_local ifds j
```

Description RFC1122 says loopback addresses must never appear on the wire. Here we test if this segment is in violation. Ideally, we'd check "(src or dest in loopback net) and interface not loopback", but we can't see which interface it's going out of in this model. The condition above is possibly the best approximation we can make if one considers the possible values of `msg.is1` and `msg.is2`.

12.3 Files, file descriptors, and sockets (TCP and UDP)

The open files of a host are modelled by a set of open file descriptions, indexed by fid . The open files of a process are identified by file descriptor fd , which is an index into a table of $fids$. This table is modelled by a finite map. File descriptors are isomorphic to the natural numbers.

12.3.1 Summary

| | |
|----------------------|-----------------------------------|
| <i>fdlt</i> | < comparison on file descriptors |
| <i>fdle</i> | ≤ comparison on file descriptors |
| <i>leastfd</i> | least fd satisfying predicate P |
| <i>nextfd</i> | next file descriptor to use |
| <i>fid_ref_count</i> | count references to given fid |
| <i>sane_socket</i> | socket sanity invariants hold |

12.3.2 Rules

– **< comparison on file descriptors :**

```
fdlt(FD n)(FD m) = n < m
```

– **≤ comparison on file descriptors :**

```
fdle(FD n)(FD m) = n ≤ m
```

– **least fd satisfying predicate P :**

```
leastfd P = FD(least n.P(FD n))
```

– **next file descriptor to use :**

```
nextfd arch fds fd' = if windows_arch arch then
  (* no ordering on Windows fds; they're just handles *)
  fd' ∉ dom(fds)
else
  (* POSIX architectures allocate in order *)
  fd' = leastfd fd'.fd' ∉ dom(fds)
```

Description Basic operations on file descriptors. Normally, when a new file descriptor is required the least unused one is used.

Variations

| | |
|-------|--|
| WinXP | On Windows, file descriptors are opaque handles, and have no useful ordering. In particular, nextfd returns an arbitrary unused file descriptor. |
|-------|--|

– **count references to given *fid* :**

$\text{fid_ref_count}(fds : fd \mapsto fid, fid) = \text{card}(\text{dom}(\text{rrestrict } fds\{fid\}))$

Description A file is closed when its reference count drops to zero. This function determines the reference count of a file (strictly, a *fid*).

– **socket sanity invariants hold :**

$\text{sane_socket } sock = \text{case } sock.pr \text{ of}$
 TCP_PROTO *tcp_sock* \rightarrow
 (*LENGTH *tcp_sock.rcvq* \leq sock.sf.n(SO_RCVBUF) \wedge (* true?? *)*)
 length *tcp_sock.rcvq* \leq TCP_MAXWIN \ll TCP_MAXWINSIZE (* \wedge *)
 (*LENGTH *tcp_sock.sndq* \leq sock.sf.n(SO_SNDBUF) (* true?? *)*)
 || UDP_PROTO *udp_sock* \rightarrow
 T

Description There are some demonstrable invariants on a socket; this definition asserts them. These are largely here to provide explicit bounds to the symbolic evaluator.

12.4 Binding (TCP and UDP)

Both TCP and UDP have a concept of a socket being *bound* to a local port, which means that that socket may receive datagrams addressed to that port. A specific local IP address may also be specified, and a remote IP address and/or port. This ‘quadruple’ (really a quintuple, since the protocol is also relevant) is used to determine the socket that best matches an incoming datagram.

The functions in this section determine this best-matching socket, using rules appropriate to each protocol. Support is also provided for determining which ports are available to be bound by a new socket, and for automatically choosing a port to bind to in cases where the user does not specify one.

12.4.1 Summary

| | |
|--------------------------------------|--|
| <i>bound_ports_protocol_autobind</i> | the set of ports currently bound by a socket for a protocol |
| <i>bound_port_allowed</i> | is it permitted to bind the given (IP,port) pair? |
| <i>autobind</i> | set of ports available for autobinding |
| <i>bound_after</i> | was <i>sid</i> bound more recently than <i>sid'</i> ? |
| <i>match_score</i> | score the match against the given pattern of the given quadruple |
| <i>lookup_udp</i> | the set of sockets matching an address quad, for UDP |
| <i>tcp_socket_best_match</i> | the set of sockets matching a quad, for TCP |
| <i>lookup_icmp</i> | the set of sockets matching a quad, for ICMP |

12.4.2 Rules

– **the set of ports currently bound by a socket for a protocol :**

```
bound_ports_protocol_autobind pr socks = {p | ∃s : socket.
                                     s ∈ rng(socks) ∧ s.ps1 = ↑ p ∧
                                     proto_of s.pr = pr}
```

Description Rebinding of ports already bound is often restricted. bound_ports_protocol_autobind is a list of all ports having a socket of the given protocol binding that port.

– **is it permitted to bind the given (IP,port) pair? :**

```
bound_port_allowed pr socks sf arch is p =
  p ∉
  {port | ∃s : socket.
   s ∈ rng(socks) ∧ s.ps1 = ↑ port ∧
   proto_eq s.pr pr ∧
   (if bsd_arch arch ∧ SO_REUSEADDR ∈ sf.b then
     s.is2 = * ∧ s.is1 = is
   else if linux_arch arch ∧ SO_REUSEADDR ∈ sf.b ∧ SO_REUSEADDR ∈ s.sf.b ∧
     ((∃tcp_sock.TCP_PROTO(tcp_sock) = s.pr ∧ ¬(tcp_sock.st = LISTEN)) ∨
      ∃udp_sock.UDP_PROTO(udp_sock) = s.pr) then
     F(* If socket is not in LISTEN state or is a UDP socket can always rebind here *)
   else if windows_arch arch ∧ SO_REUSEADDR ∈ sf.b then
     F(* can rebind any UDP address; not sure about TCP - assume the same for now *)
   else
     (is = * ∨ s.is1 = * ∨ (∃i : ip.is = ↑ i ∧ s.is1 = ↑ i))}}
```

Description This determines whether binding a socket (of protocol *pr*) to local address *is*, *p* is permitted, by considering the other bound sockets on the host and the state of the sockets' SO_REUSEADDR flags. Note: SB believes this definition is correct for TCP and UDP on BSD and Linux through exhaustive manual verification. Note: WinXP is still to be checked.

– **set of ports available for autobinding :**

```
autobind(↑ p, -, -) = {p} ∧
autobind(*, pr, socks) = ephemeral_ports diff(bound_ports_protocol_autobind pr socks)
```

Description Note that SO_REUSEADDR is not considered when choosing a port to autobind to.

– **was sid bound more recently than sid'? :**

```
bound_after sid sid'[] = ASSERTION_FAILURE“bound_after”(* should never reach this case *) ∧
bound_after sid sid'(sid0 :: bound) =
if sid = sid0 then T(* newly-bound sockets are added to the head *)
else if sid' = sid0 then F
  else bound_after sid sid' bound
```

– **score the match against the given pattern of the given quadruple :**

```
(match_score(-, *, -, -) = 0 $n$ ) ∧
```

```

(match_score(*, ↑ p1, *, *) (i3, ps3, i4, ps4) =
  if ps4 = ↑ p1 then 1 else 0) ∧
(match_score(↑ i1, ↑ p1, *, *) (i3, ps3, i4, ps4) =
  if (i1 = i4) ∧ (↑ p1 = ps4) then 2 else 0) ∧
(match_score(↑ i1, ↑ p1, ↑ i2, *) (i3, ps3, i4, ps4) =
  if (i2 = i3) ∧ (i1 = i4) ∧ (↑ p1 = ps4) then 3 else 0) ∧
(match_score(↑ i1, ↑ p1, ↑ i2, ↑ p2) (i3, ps3, i4, ps4) =
  if (↑ p2 = ps3) ∧ (i2 = i3) ∧ (i1 = i4) ∧ (↑ p1 = ps4) then 4
  else 0)

```

Description These two functions are used to match an incoming UDP datagram to a socket. The `bound_after` function returns **T** if the socket `sid` (the first argument) was bound after the socket `sid'` (the second argument) according to a list of bound sockets (the third argument).

The `match_score` function gives a score specifying how closely two address quads, one from a socket and one from a datagram, correspond; a higher score indicates a more specific match.

– the set of sockets matching an address quad, for UDP :

`lookup_udp socks quad bound arch =`

`{sid | sid ∈ dom(socks) ∧`

`let s = socks[sid] in`

`let sn = match_score(s.is1, s.ps1, s.is2, s.ps2)quad in`

`sn > 0 ∧`

`if windows_arch arch then`

`if sn = 1 then`

`¬(∃(sid', s') :: (socks \ sid). match_score(s'.is1, s'.ps1, s'.is2, s'.ps2)quad > sn)`

`else T`

`else`

`¬(∃(sid', s') :: (socks \ sid).`

`(match_score(s'.is1, s'.ps1, s'.is2, s'.ps2)quad > sn ∨`

`(linux_arch arch ∧ match_score(s'.is1, s'.ps1, s'.is2, s'.ps2)quad = sn ∧`

`bound_after sid' sid bound))))}`

Description This function returns a set of UDP sockets which the datagram with address quad `quad` may be delivered to. For FreeBSD and Linux there is only one such socket; for WinXP there may be multiple.

For each socket in the finite map of sockets `socks`, the score, `sn`, of the matching of the socket's address quad and `quad` is computed using `match_score` (p??).

Variations

| | |
|---------|---|
| FreeBSD | For FreeBSD, the set contains the sockets for which the score is greater than zero and there is no other socket in <code>socks</code> with a higher score. |
| Linux | For Linux, the set contains the sockets for which the score is greater than zero, there are no sockets with a higher score, and the socket was bound to its local port after all the other sockets with the same score. |
| WinXP | For WinXP, the set contains all the sockets with score greater than one and also the sockets for which the score is one, <code>sn = 1</code> , and there are no sockets with greater scores. |

– **the set of sockets matching a quad, for TCP :**

```
tcp_socket_best_match(socks : sid ↦ socket)(sid, sock)(seg : tcpSegment) arch =
(* is the socket sid the best match for segment seg? *)
let s = sock in
let score = match_score(s.is1, s.ps1, s.is2, s.ps2)
                (the seg.is1, seg.ps1, the seg.is2, seg.ps2) in
¬(∃(sid', s') :: socks \ sid.
    match_score(s'.is1, s'.ps1, s'.is2, s'.ps2)
        (the seg.is1, seg.ps1, the seg.is2, seg.ps2) > score)
```

Description This function determines whether a given socket `sid` is the best match for a received TCP segment `seg`.

The score (obtained using `match_score` (p??)) for the given socket is determined, and compared with the score for each other socket in `socks`. If none have a greater score, this is the best match and `true` is returned; otherwise, `false` is returned.

– **the set of sockets matching a quad, for ICMP :**

```
lookup_icmp socks icmp arch bound =
{sid0 | ∃(sid, sock) :: socks.
    sock.ps1 = icmp.ps3 ∧ proto_of sock.pr = icmp.proto ∧ sid0 = sid ∧
    if windows_arch arch then T
    else
        sock.is1 = icmp.is3 ∧ sock.is2 = icmp.is4 ∧
        (sock.ps2 = icmp.ps4 ∨
        (linux_arch arch ∧
            proto_of sock.pr = PROTO_UDP ∧ sock.ps2 = * ∧
            ¬(∃(sid', s) :: (socks \ sid).
                s.is1 = icmp.is3 ∧ s.is2 = icmp.is4 ∧
                s.ps1 = icmp.ps3 ∧ s.ps2 = icmp.ps4 ∧
                proto_of s.pr = icmp.proto ∧
                bound_after sid' sid bound)
            )))
}}
```

Description

This function returns the set of sockets matching a received ICMP datagram `icmp`.

An ICMP datagram contains the initial portion of the header of the original message to which it is a response. For a socket to match, it must at least be bound to the same port and protocol as the source of the original message. Beyond this, architectures differ. Usually, the socket must be connected, and connected to the same port as the original destination; and the source and destination IP addresses must agree.

Variations

| | |
|---------|---|
| WinXP | For Windows, the socket need not be connected, and the source and destination IP addresses need not agree; an ICMP is delivered to one socket bound to the same port and protocol as the original source. |
| Linux | For Linux, UDP ICMPs may also be delivered to unconnected sockets, as long as no matching connected socket was bound more recently than that socket. |
| FreeBSD | For FreeBSD, the behaviour is as described above. |

12.5 Timers (TCP and UDP)

Many TCP protocol events are time-dependent, and time is also necessary for a useful specification of the behaviour of system calls, returns, and datagram emission and receipt. These common time-dependent behaviours are described using the timers below.

12.5.1 Summary

| | |
|-----------------------|--|
| <i>slow_timer</i> | TCP slow timer, typically 500ms resolution (for keepalive, MSL, linger, badrxtwin) |
| <i>fast_timer</i> | TCP fast timer, typically 200ms resolution (for delack) |
| <i>kern_timer</i> | kernel timer, typically 10ms resolution (for timestamp valid, pselect) |
| <i>sched_timer</i> | scheduling timer (for OS returns) |
| <i>inqueue_timer</i> | in-queue timer (incoming message processing) |
| <i>outqueue_timer</i> | out-queue timer (outgoing message emission) |

12.5.2 Rules

-
- **TCP slow timer, typically 500ms resolution (for keepalive, MSL, linger, badrxtwin) :**
`slow_timer d = fuzzy_timer d SLOW_TIMER_INTVL SLOW_TIMER_MODEL_INTVL`
 - **TCP fast timer, typically 200ms resolution (for delack) :**
`fast_timer d = fuzzy_timer d FAST_TIMER_INTVL FAST_TIMER_MODEL_INTVL`
 - **kernel timer, typically 10ms resolution (for timestamp valid, pselect) :**
`kern_timer d = fuzzy_timer d KERN_TIMER_INTVL KERN_TIMER_MODEL_INTVL`
 - **scheduling timer (for OS returns) :**
`sched_timer = upper_timer dschedmax`
 - **in-queue timer (incoming message processing) :**
`inqueue_timer = upper_timer diqmax`
 - **out-queue timer (outgoing message emission) :**
`outqueue_timer = upper_timer doqmax`
-

Description

Traditionally TCP has been implemented using two timers, a slow timer ticking once every 500ms, and a fast timer ticking once every 200ms. In addition, the kernel is assumed to maintain a tick count, typically incremented every 10ms.

Measuring intervals with such a timer means an uncertainty in duration: the observed interval may be up to one tick less than the specified interval, and is on average half a tick less. We model this with a `fuzzy_timer` (p47), fuzzy to the left by *eps* and to the right by *fuz*, i.e., $[d - eps, d + fuz]$.

The *eps*, one tick, accounts for the fact that we do not know where in the clock's period we set the timer.

The *fuz* (some global fuzziness) is included to account for the atomicity of the model. For example, an implementation TCP processing step, performed by `tcp_output` etc., occupies some time interval, with timers such as `tt_rexmt` being reset at various points within that interval. The model, on the other hand, has atomic transitions. The possible time difference between multiple timer resets in the same step must be accounted for by this fuzziness.

For example, a model rule may reset the `tt_rexmt` timer and also leave a segment on the output queue, with time passing before the segment is seen on the wire.

The various flavours of `upper_timer` (`p??`) – `sched_timer`, `inqueue_timer`, `outqueue_timer` – fire at any time between now and *dmax*. These events may occur at any time up to a specified maximum delay.

12.6 Time values for socket options (TCP and UDP)

The TLang sockets interface representation of a time is as a pair of integers, the first for seconds and the second for nanoseconds. It also uses (int#int) option representations, e.g. in the arguments to `setsockopt` and `pselect` and the result of `setsockopt`, with the *None* value meaning infinity. Internally, time is represented as a time value, either a real or infinity. These routines convert between the various types. Note that they allow ill-formed `tltimeopts` without complaint.

12.6.1 Summary

| | |
|--------------------------------|---|
| <code>time_of_tlname</code> | convert (<i>sec, nsec</i>) pair to real time value |
| <code>time_of_tlnameopt</code> | convert optional (<i>sec, nsec</i>) pair to real time value (where * mapped to ∞) |
| <code>tltimeopt_wf</code> | is an optional (<i>sec, nsec</i>) pair well-formed? |
| <code>tltimeopt_of_time</code> | convert a time value to an optional (<i>sec, nsec</i>) pair |

12.6.2 Rules

-
- **convert (*sec, nsec*) pair to real time value :**
 $(\text{time_of_tltime} : \text{int}\#\text{int} \rightarrow \text{time})$
 $(\text{sec}, \text{nsec}) = \text{time}(\text{real_of_int } \text{sec} + \text{real_of_int } \text{nsec}/1000000000)$
 - **convert optional (*sec, nsec*) pair to real time value (where * mapped to ∞) :**
 $\text{time_of_tltimeopt } * = \infty \wedge$
 $\text{time_of_tltimeopt}(\uparrow \text{sn}) = \text{time_of_tltime } \text{sn}$
 - **is an optional (*sec, nsec*) pair well-formed? :**
 $(\text{tltimeopt_wf} : (\text{int}\#\text{int}) \text{option} \rightarrow \text{bool})$
 $* = \mathbf{T} \wedge$
 $\text{tltimeopt_wf}(\uparrow(\text{sec}, \text{nsec})) = (\text{sec} \geq 0 \wedge \text{nsec} \geq 0 \wedge \text{nsec} < 1000000000)$
 - **convert a time value to an optional (*sec, nsec*) pair :**
 $(\text{tltimeopt_of_time} : \text{time} \rightarrow (\text{int}\#\text{int}) \text{option})t$
 $= @x.\text{tltimeopt_wf } x \wedge \text{time_of_tltimeopt } x = t$ (* garbage if *t* not nonnegative integral number of nsec *)
-

Description A *tltimeopt* is well-formed if *sec* and *nsec* are positive and *nsec* is less than 10^9 .

12.7 Queues (TCP and UDP)

Messages are queued at various points within the implementations, e.g. within the network interface hardware and in the kernel. These queues can become full, though their "size" is not simple to describe — e.g. in BSD there is some accounting of the number of mbufs used. We model this with simple queues, for example the host message inqueue and outqueue (see *iq* and *oq*, [host \(p61\)](#)) which have lists of messages. These model the combination of network interface and kernel queues. We allow them to nondeterministically be full for enqueue operations, to ensure that the specification includes all real-world traces. This behaviour is guarded by `INFINITE_RESOURCES`.

The nondeterminism means that queue operations must be relations, not functions, and hence that many definitions that use them must also be relational.

Many queues also associated with timers (see e.g. `inqueue_timer (p??)`) bounding the times within which they must next be processed.

One might want additional properties, e.g. (1) if a queue is empty then at least one message can be enqueued, or more generally a specified finite lower bound on queue size; or (2) if a queue is full then it remains so until a message is dequeued (perhaps only for enqueue attempts of at least the same size). At present we see no need for the additional complication.

12.7.1 Summary

| | |
|------------------------------|--|
| <i>enqueue</i> | attempt to enqueue a message |
| <i>enqueue_iq</i> | attempt to enqueue onto the in-queue |
| <i>enqueue_oq</i> | attempt to enqueue onto the out-queue |
| <i>dequeue</i> | attempt to dequeue a message |
| <i>dequeue_iq</i> | attempt to dequeue from the in-queue |
| <i>dequeue_oq</i> | attempt to dequeue from the out-queue |
| <i>route_and_enqueue_oq</i> | attempt to route and then enqueue an outgoing message |
| <i>enqueue_list_qinfo</i> | attempt to enqueue a list of messages |
| <i>enqueue_list</i> | attempt to enqueue a list of messages, ignoring success flags |
| <i>enqueue_oq_list_qinfo</i> | attempt to enqueue a list of messages onto the out-queue |
| <i>enqueue_oq_list</i> | attempt to enqueue a list of messages onto the out-queue, ignoring success flags |
| <i>accept_incoming_q0</i> | should an incoming incomplete connection be accepted? |
| <i>accept_incoming_q</i> | should an incoming completed connection be accepted? |
| <i>drop_from_q0</i> | drop from incomplete-connection queue? |

12.7.2 Rules

– **attempt to enqueue a message :**
 $enqueue\ dq((q)_d, msg, (q')_{d'}, queued)$
 $= ((INFINITE_RESOURCES \implies queued) \wedge$
 $(q', d') = (\mathbf{if}\ queued\ \mathbf{then}\ (q\ @\ [msg], dq)\ \mathbf{else}\ (q, d))$
 $)$

Description This is a relation between an original timed queue $(q)_d$, a message to enqueue, msg , a resulting timed queue $(q')_{d'}$, and a boolean $queued$ indicating whether the enqueue was successful or not. For a successful enqueue the timer on the resulting queue is set to dq

– **attempt to enqueue onto the in-queue :**
 $enqueue_iq = enqueue\ inqueue_timer$

– **attempt to enqueue onto the out-queue :**
 $enqueue_oq = enqueue\ outqueue_timer$

Description Add a message to the respective queue, returning the new queue and a flag saying whether the message was successfully queued.

– **attempt to dequeue a message :**
 $dequeue\ dq((q)_d, (q')_{d'}, msg)$
 $= \mathbf{case}\ q\ \mathbf{of}$
 $(msg0 :: q_0) \rightarrow q' = q_0 \wedge msg = \uparrow\ msg0 \wedge d' = (\mathbf{if}\ q_0 = []\ \mathbf{then}\ never_timer\ \mathbf{else}\ dq) \parallel$
 $[] \rightarrow q' = q \wedge msg = * \wedge d' = d$

– **attempt to dequeue from the in-queue :**
 $dequeue_iq = dequeue\ inqueue_timer$

– **attempt to dequeue from the out-queue :**
 $dequeue_oq = dequeue\ outqueue_timer$

Description Remove a message from the queue, returning the new queue, and the message if there is one.

– **attempt to route and then enqueue an outgoing message :**

```
route_and_enqueue_oq(rttab, ifds, oq, msg, oq', es, arch)
= case test_outroute(msg, rttab, ifds, arch) of
  * → F
|| ↑(↑ e) → oq' = oq ∧ es = ↑ e
|| ↑ * → ∃queued.
      enqueue_oq(oq, msg, oq', queued) ∧
      es = if queued then * else ↑ ENOBUFS
```

Description This is a relation because enqueue_oq can non-deterministically decide that the *oq* is full.

– **attempt to enqueue a list of messages :**

```
enqueue_list_qinfo dq(q, (msg, queued) :: msgqs, q')
= (∃q0.
  enqueue dq(q, msg, q0, queued) ∧
  enqueue_list_qinfo dq(q0, msgqs, q') ∧
  enqueue_list_qinfo dq(q, [], q')
= (q' = q)
```

– **attempt to enqueue a list of messages, ignoring success flags :**

```
enqueue_list dq(q, msgqs, q', queued) =
(∃msgqs.
  enqueue_list_qinfo dq(q, msgqs, q') ∧
  msgqs = map fst msgqs ∧
  queued = every(λx. snd x = T)msgqs)
```

– **attempt to enqueue a list of messages onto the out-queue :**

```
enqueue_oq_list_qinfo = enqueue_list_qinfo outqueue_timer
```

– **attempt to enqueue a list of messages onto the out-queue, ignoring success flags :**

```
enqueue_oq_list = enqueue_list outqueue_timer
```

Description We sometimes need to enqueue multiple messages at a time. enqueue_list_qinfo tries to enqueue a list of messages, pairing each with its success boolean.

Often, we don't care too much about the precise queueing success of each message. enqueue_list provides the AND of success of each message (though this is of limited use).

– **should an incoming incomplete connection be accepted? :**

```
accept_incoming_q0(lis : socket_listen)(b : bool)
= (b = length lis.q < backlog_fudge lis.qlimit)
```

– **should an incoming completed connection be accepted? :**

```
accept_incoming_q(lis : socket_listen)(b : bool)
= (b = length lis.q < 3 * backlog_fudge lis.qlimit div 2)
```

– **drop from incomplete-connection queue? :**

```
drop_from_q0(lis : socket_listen)(b : bool)
= ((length lis.q0 ≥ TCP_Q0MINLIMIT ∧ b = T) ∨
  (length lis.q0 < TCP_Q0MAXLIMIT ∧ b = F))
```

Description A listening socket has two queues, the incomplete connections queue *lis.q₀* and the completed connections queue *lis.q*. An incoming incomplete (respectively, completed) connection be accepted onto *lis.q₀* (respectively, *lis.q*) if the relevant queue is not full. Intriguingly, for FreeBSD 4.6-RELEASE, this specification is correct, but if syncaches were to be turned off, the condition in the *q₀* case would be **length** *lis.q* < 3 * *lis qlimit*/2 instead. Existing incomplete connections may dropped from *lis.q₀* to make room if its length is between its minimum and maximum limits.

12.8 TCP Options (TCP only)

TCP option handling.

12.8.1 Summary

| | |
|----------------------------------|--|
| <i>do_tcp_options</i> | Constrain the TCP timestamp option values that appear in an outgoing segment |
| <i>calculate_tcp_options_len</i> | Calculate the length consumed by the TCP options in a real TCP segment |

12.8.2 Rules

– **Constrain the TCP timestamp option values that appear in an outgoing segment :**

```
do_tcp_options cb_tf_doing_tstmp cb_ts_recent cb_ts_val =
if cb_tf_doing_tstmp then
  let ts_ecr' = option_case (ts_seq 0w) I (timewindow_val_of cb_ts_recent) in
    ↑(cb_ts_val, ts_ecr')
else
  *
```

– **Calculate the length consumed by the TCP options in a real TCP segment :**

```
calculate_tcp_options_len cb_tf_doing_tstmp =
if cb_tf_doing_tstmp then 12 else 0 : num
```

Description This calculation omits window-scaling and mss options as these only appear in SYN segments during connection setup. The total length consumed by all options will always be a multiple of 4 bytes due to padding. If more TCP options were added to the model, the space consumed by options would be architecture/options/alignment/padding dependent.

12.9 Buffers, windows, and queues (TCP and UDP)

Various functions that compute buffer sizes, window sizes, and remaining send queue space. Some of these computations are architecture-specific.

12.9.1 Summary

| | |
|----------------------------|--|
| <i>calculate_buf_sizes</i> | Calculate buffer sizes for <i>rcvbufsize</i> , <i>sndbufsize</i> , <i>t_maxseg</i> , and <i>snd_cwnd</i> |
| <i>calculatebsdrcvwnd</i> | Calculation of <i>rcvwnd</i> |
| <i>send_queue_space</i> | Rule version: \$Id: TCP1_auxFnsScript.sml,v 1.219 2005/03/17 11:35:34 kw217 Exp \$ |

12.9.2 Rules

– **Calculate buffer sizes for *rcvbufsize*, *sndbufsize*, *t_maxseg*, and *snd_cwnd* :**

```

calculate_buf_sizes cb_t_maxseg seg_mss bw_delay_product_for_rt is_local_conn
    rcvbufsize sndbufsize cb_tf_doing_tstamp arch =

let t_maxseg' =
  (* TCPv2p901 claims min 32 for "sanity"; FreeBSD4.6 has 64 in tcp_mss(). BSD has the route MTU if avail, or
  min MSSDFLT(link MTU) otherwise, as the first argument of the MIN below. That is the same calculation as we
  did in connect_1. We don't repeat it, but use the cached value in cb.t_maxseg. *)
let maxseg = (min cb_t_maxseg(max 64(option_case MSSDFLT I seg_mss))) in
  if linux_arch arch then
    maxseg
  else
    (* BSD subtracts the size consumed by options in the TCP header post connection establishment. The WinXP
    and Linux behaviour has not been fully tested but it appears Linux does not do this and WinXP does. *)
    maxseg - (calculate_tcp_options_len cb_tf_doing_tstamp)

in
  (* round down to multiple of cluster size if larger (as BSD). From BSD code; assuming true for WinXP for now *)
let t_maxseg'' = if linux_arch arch then t_maxseg'(* from tests *)
  else roundup MCLBYTES t_maxseg' in

(* buffootle: rcv *)
let rcvbufsize' = option_case rcvbufsize I bw_delay_product_for_rt in
let (rcvbufsize'', t_maxseg''') = (if rcvbufsize' < t_maxseg''
  then (rcvbufsize', rcvbufsize')
  else (min SB_MAX(roundup t_maxseg'' rcvbufsize'),
  t_maxseg''')) in

(* buffootle: snd *)
let sndbufsize' = option_case sndbufsize I bw_delay_product_for_rt in
let sndbufsize'' = (if sndbufsize' < t_maxseg''
  then sndbufsize'
  else min SB_MAX(roundup t_maxseg'' sndbufsize')) in

(* compute initial cwnd *)
let snd_cwnd = t_maxseg'' * (if is_local_conn then SS_FLTSZ_LOCAL else SS_FLTSZ) in
(rcvbufsize'', sndbufsize'', t_maxseg''', snd_cwnd)

```

Description Used in *deliver_in_1* and *deliver_in_2*.

– **Calculation of *rcv_wnd* :**

```

calculate_bsd_rcv_wnd sf tcp_sock =
max(num(tcp_sock.cb.rcv_adv - tcp_sock.cb.rcv_next))
  (sf.n(SO_RCVBUF) - length tcp_sock.rcvq)

```

Description Calculation of *rcv_wnd* as done in BSD's `tcp_input.c`, line 1052. The model currently calls this from `tcp_output_really` in post-ESTABLISHED states, using *deliver_in_3* to update *rcv_wnd* as soon as a segment comes, rather than waiting for the next *deliver_in*, as BSD does — this is a saner thing to do. In order to comply with BSD however, we need *calculate_bsd_rcv* to be called on receipt of the first 'real' (i.e. non-syncache) segment, to update *rcv_wnd* from the temporary initial value.

```

- :
send_queue_space(sndq_max : num) sndq_size oob arch maxseg i2 =
  {n | if bsd_arch arch then
    n ≤ (sndq_max - sndq_size) + (if oob then oob_extra_sndbuf else 0)
  else if linux_arch arch then
    (if in_loopback i2 then
      n = maxseg + ((sndq_max - sndq_size) div 16816) * maxseg
    else
      n = (2 * maxseg) + ((sndq_max - sndq_size - 1890) div 1888) * maxseg)
  else n ≥ 0}

```

Description Calculation of the usable send queue space.

FreeBSD calculates send buffer space based on the byte-count size and max, and the number and max of mbufs. As we do not model mbuf usage precisely we are somewhat nondeterministic here.

Linux calculates it based on the MSS: the space is some multiple of the MSS; the number of bytes for each MSS-sized segment is the MSS+overhead where overhead is 420+(20 if using IP), which is why the `i2` argument is needed.

Windows is very strange. Leaving it completely unconstrained is not what actually happens, but more investigation is needed in future to determine the actual behaviour.

12.10 Band limiting (TCP and UDP)

The rate of emission of certain TCP and ICMP responses from a host is often controlled by a bandwidth limiter. This limits resource usage in the event of some error conditions, and also defends against certain denial-of-service attacks.

Responses that may be bandlimited are grouped into categories (`bandlim_reason`), and bandlimiting is applied to each category separately. Bandlimiting is applied across the entire host, not per socket or process. There are a range of different schemes that may be used, from none at all, through limiting the number of packets in any given second, to a decaying average tuned to limit bursts and sustained throughput differently. We provide specifications for the first two.

12.10.1 Summary

| | |
|------------------------------------|--|
| <code>bandlim_state_init</code> | initial state of bandlimiter |
| <code>bandlim_rst_ok_always</code> | the trivial 'always OK' bandlimiter |
| <code>simple_limit</code> | simple-bandlimiter rate settings |
| <code>bandlim_rst_ok_simple</code> | a simple rate-limiting bandlimiter |
| <code>bandlim_rst_ok</code> | the bandlimiter actually used |
| <code>enqueue_oq_bndlim_rst</code> | enqueue onto out-queue if allowed by bandlimiter |

12.10.2 Rules

– **initial state of bandlimiter :**

```
bandlim_state_init = [] : bandlim_state
```

– **the trivial 'always OK' bandlimiter :**

```
(bandlim_rst_ok_always : tcpSegment# ts_seq #bandlim_reason#bandlim_state → bool#bandlim_state)
(seg, ticks, reason, bndlm)
= let bndlm' = (seg, ticks, reason) :: bndlm
  in
  (T, bndlm')
```

– **simple-bandlimiter rate settings :**

```
(simple_limit : bandlim_reason → num option)
  BANDLIM_UNLIMITED = * ∧
simple_limit BANDLIM_RST_CLOSEDPORT = ↑ 200 ∧
simple_limit BANDLIM_RST_OPENPORT = ↑ 200
– a simple rate-limiting bandlimiter :
(bandlim_rst_ok_simple : tcpSegment# ts_seq #bandlim_reason#bandlim_state → bool#bandlim_state)
  (seg, ticks, reason, bndlm)
= let reasoneq = (λr₀.λ(s, t, r).r = r₀)
  and ticksgt = (λt₀.λ(s, t, r).t > t₀)
  in
  let count = length(filter(reasoneq reason)(TAKEWHILE(ticksgt(ticks – num_floor(1 * HZ)))bndlm))
  in
  ((case simple_limit reason of
    * → T
    || ↑ n → count < n),
  (seg, ticks, reason) :: bndlm)
```

Description Simple bandlimiter: limit number of ICMPs in the last second to the listed value. This is based roughly on the BSD behaviour, save that for BSD it is "since the last second" not "in the last second".

– **the bandlimiter actually used :**
bandlim_rst_ok = bandlim_rst_ok_simple

Description Which band limiter to use?

– **enqueue onto out-queue if allowed by bandlimiter :**
enqueue_oq_bndlim_rst(oq, seg, ticks, reason, bndlm, oq', bndlm', queued_or_dropped)
= let (emit, bndlm₀) = bandlim_rst_ok(seg, ticks, reason, bndlm)
 in
 bndlm' = bndlm₀ ∧
 if emit then
 enqueue_oq(oq, TCP seg, oq', queued_or_dropped)
 else
 (oq' = oq ∧ queued_or_dropped = **T**)

Description For convenience, combine enqueueing and bandlimiting into a single function.

12.11 UDP support (UDP only)

Performing a UDP send, filling in required details as necessary.

12.11.1 Summary

dosend

do a UDP send, filling in source address and port as necessary

12.11.2 Rules

– **do a UDP send, filling in source address and port as necessary :**

```
(dosend(ifds, rfttab, (*, data), (↑ i1, ↑ p1, ↑ i2, ps2), oq, oq', ok) =
enqueue_oq(oq, UDP(⟦ is1 := ↑ i1; is2 := ↑ i2;
                    ps1 := ↑ p1; ps2 := ps2;
                    data := data⟧),
            oq', ok) ∧
(dosend(ifds, rfttab, (↑(i, p), data), (*, ↑ p1, *, *), oq, oq', ok) =
(∃ i'1. enqueue_oq(oq, UDP(⟦ is1 := ↑ i'1; is2 := ↑ i;
                            ps1 := ↑ p1; ps2 := ↑ p;
                            data := data⟧),
                oq', ok) ∧ i'1 ∈ auto_outroute(i, *, rfttab, ifds))) ∧
(dosend(ifds, rfttab, (↑(i, p), data), (↑ i1, ↑ p1, is2, ps2), oq, oq', ok) =
enqueue_oq(oq, UDP(⟦ is1 := ↑ i1; is2 := ↑ i;
                    ps1 := ↑ p1; ps2 := ↑ p;
                    data := data⟧),
            oq', ok))
```

Description For use in UDP *sendto*().

12.12 TCP timing and RTT (TCP only)

TCP performs repeated transmissions in three situations: retransmission of unacknowledged data, retransmission of an unacknowledged SYN, and probing a closed window ('persisting'). In each case the interval between transmissions is a function of the estimated round-trip time for the connection, and is exponentially backed off if no response is received. The RTT estimate indicates when TCP should expect a reply, and the exponential backoff controls TCP's resource usage.

12.12.1 Summary

| | |
|---------------------------|--|
| <i>tcp_backoffs</i> | select this architecture's retransmit backoff list |
| <i>tcp_syn_backoffs</i> | select this architecture's SYN-retransmit backoff list |
| <i>mode_of</i> | obtain the mode of a backoff timer |
| <i>shift_of</i> | obtain the shift of a backoff timer |
| <i>computed_rto</i> | compute retransmit timeout to use |
| <i>computed_rxtcur</i> | compute the last-used <i>rxtcur</i> |
| <i>start_tt_rexmt_gen</i> | construct retransmit timer (generic) |
| <i>start_tt_rexmt</i> | construct normal retransmit timer |
| <i>start_tt_rexmtsyn</i> | construct SYN-retransmit timer |
| <i>start_tt_persist</i> | construct persist timer |
| <i>update_rtt</i> | update RTT estimators from new measurement |
| <i>expand_cwnd</i> | expand congestion window |

12.12.2 Rules

– **select this architecture's retransmit backoff list :**

```
tcp_backoffs(arch : arch) =
if bsd_arch arch then TCP_BSD_BACKOFFS
else if linux_arch arch then TCP_LINUX_BACKOFFS
else if windows_arch arch then TCP_WINXP_BACKOFFS
else TCP_BSD_BACKOFFS (* default to BSD *)
```

```

– select this architecture's SYN-retransmit backoff list :
tcp_syn_backoffs(arch : arch) =
if bsd_arch arch then TCP_SYN_BSD_BACKOFFS
else if linux_arch arch then TCP_SYN_LINUX_BACKOFFS
else if windows_arch arch then TCP_SYN_WINXP_BACKOFFS
else TCP_SYN_BSD_BACKOFFS (* default to BSD *)

```

```

– obtain the mode of a backoff timer :
(mode_of : (rexmtmode#num)timed option → rexmtmode option)
  (↑(((mode, -))) = ↑ mode ∧
mode_of * = *

```

```

– obtain the shift of a backoff timer :
shift_of(↑(((-, shift))) = shift

```

Description TCP exponential-backoff timers are represented as (rexmtmode#num)timed option, where *mode* : rexmtmode is the current TCP output mode (see rexmtmode (p55)), and *shift* : num is the 0-origin index into the backoff list of the interval *currently underway*.

```

– compute retransmit timeout to use :
computed_rto(backoffs : num list)(shift : num)(ri : rttinf)
= real_of_num(EL shift backoffs) *
max ri.t_rttmin(ri.t_srvt + 4 * ri.t_rttvar)
– compute the last-used rxtecur :
computed_rxtecur(ri : rttinf)(arch : arch)
= max ri.t_rttmin
  (min(the TCPTV_REXMTMAX
    (computed_rto(if ri.t_wassyn then tcp_syn_backoffs arch
      else tcp_backoffs arch)
      ri.t_lastshift ri))

```

Description

computed_rto computes the retransmit timeout to be used, from the backoff list, the shift, and the current RTT estimators. The base time is $RTT + 4RTTVAR$; this is clipped against a minimum value, and then multiplied by the value from the backoff list.

computed_rxtecur is not used in constructing timers, but *tcp_output* uses it to check if TCP has been idle for a while (causing slow start to be entered again). It is an approximation to the value actually used below. Note it might be possible to make this precise rather than an approximation; also, *computed_rxtecur* and *start_tt_rexmt_gen* could be merged.

Note: TCPTV_REXMTMAX had better not be infinite!

```

– construct retransmit timer (generic) :
start_tt_rexmt_gen(mode : rexmtmode)(backoffs : num list)(shift : num)(wantmin : bool)(ri : rttinf)
= let rxtecur = max(if wantmin
  then max ri.t_rttmin(ri.t_lastrtt + 2/ HZ)
  else ri.t_rttmin)
  (min(the TCPTV_REXMTMAX (* better not be infinite! *)
    (computed_rto backoffs shift ri)
  )
)

```

```

in
↑(((mode, shift))slow_timer(time rxtcur))
– construct normal retransmit timer :
start_tt_rexmt(arch : arch) = start_tt_rexmt_gen REXMT(tcp_backoffs arch)
– construct SYN-retransmit timer :
start_tt_rexmtsyn(arch : arch) = start_tt_rexmt_gen REXMTSYN(tcp_syn_backoffs arch)
– construct persist timer :
start_tt_persist(shift : num)(ri : rtinf)(arch : arch)
= let cur = max(the TCPTV_PERSMIN (* better not be infinite! *))
      (min(the TCPTV_PERSMAX (* better not be infinite! *))
        (computed_rto(tcp_backoffs arch) shift ri)
      )
in
↑(((PERSIST, shift))slow_timer(time cur))

```

Description

Starting the retransmit, *SYN*-retransmit, and persist timers: these function return the new timer with the given shift. This models both initialisation on receiving a segment, and update in the retransmit timer handler.

There are two alternative clipping values used for the minimum timer. *ri.t_rttmin* is used always, but in one place *t.last_rtt + 2/HZ* (i.e., 0.02s plus the last measured RTT) is used as well. The BSD sources have a comment here saying "minimum feasible timer"; it is a puzzle why this value is not used elsewhere also. (tcp_input.c:2408 vs tcp_timer.c:394, tcp_input.c:2542).

Starting the persist timer is similar to starting the retransmit timers, but the bounds are different.

Note that we don't need to look at *tf_srttvalid*, since in any case *t_srtt* and *t_rttvar* will have sensible values. That flag is just for the benefit of *update_rtt*.

```

– update RTT estimators from new measurement :
update_rtt(rtt : duration)(ri : rtinf)
= let (t_srtt', t_rttvar')
  = (if ri.tf_srtt_valid then
    let delta = (rtt - 1/HZ) - ri.t_srtt
      in
      let vardelta = abs delta - ri.t_rttvar
        in
        let t_srtt' = max(1/(32 * HZ))(ri.t_srtt + (1/8) * delta)
          and t_rttvar' = max(1/(16 * HZ))(ri.t_rttvar + (1/4) * vardelta)
            (* BSD behaviour is never to let these go to zero, but clip at the least positive value. Since SRTT
              is measured in 1/32 tick and RTTVAR in 1/16 tick, these are the minimum values. A more natural
              implementation would clip these to zero. *)
          in
          (t_srtt', t_rttvar')
    else
      let t_srtt' = rtt
        and t_rttvar' = rtt/2
          in
          (t_srtt', t_rttvar'))
in
ri ( t_rttupdated := ri.t_rttupdated + 1;
    tf_srtt_valid := T;
    t_srtt := t_srtt';
    t_rttvar := t_rttvar';
    t_lastrtt := rtt;
    t_lastshift := 0;
    t_wassyn := F(* if t_lastshift=0, this doesn't make a difference *)

```

```
(* t_softerror, t_rttseg, and t_rxtcur must be handled by the caller *)
}
```

Description Update the round trip time estimators on obtaining a new instantaneous value. Based on a close reading of `tcp_xmit_timer()`, `tcp_input.c:2347-2419`.

– **expand congestion window :**

```
expand_cwnd ssthresh maxseg maxwin cwnd
= min maxwin(cwnd + (if cwnd > ssthresh then (maxseg * maxseg) div cwnd else maxseg))
```

Description

Congestion window expansion is linear or exponential depending on the current threshold `ssthresh`.

12.13 Path MTU Discovery (TCP only)

For efficiency and reliability, it is best to send datagrams that do not need to be fragmented in the network. However, TCP has direct access only to the maximum packet size (MTU) for the interfaces at either end of the connection – it has no information about routers and links in between.

To determine the MTU for the entire path, TCP marks all datagrams ‘do not fragment’. It begins by sending a large datagram; if it receives a ‘fragmentation needed’ ICMP in return it reduces the size of the datagram and repeats the process. Most modern routers include the link MTU in the ICMP message; if the message does not contain an MTU, however, TCP uses the next lower MTU in the table below.

12.13.1 Summary

| | |
|---------------------------|------------------------------------|
| <code>next_smaller</code> | find next-smaller element of a set |
| <code>mtu_tab</code> | path MTU plateaus to try |

12.13.2 Rules

– **find next-smaller element of a set :**

```
(next_smaller : (num → bool) → num → num)  $xs\ y = @x :: xs.x < y \wedge \forall x' :: xs.x' > x \implies x' \geq y$ 
```

– **path MTU plateaus to try :**

```
mtu_tab arch = if linux_arch arch then
  {32000; 17914; 8166; 4352; 2002; 1492; 576; 296; 216; 128; 68} : num set
else
  {65535; 32000; 17914; 8166; 4352; 2002; 1492; 1006; 508; 296; 68}
```

Description MTUs to guess for path MTU discovery. This table is from RFC1191, and is the one that appears in BSD.

On `comp.protocols.tcp-ip`, Sun, 15 Feb 2004 01:38:26 -0000, <102tj-cifv6vgm02@corp.supernews.com>, `kml@bayarea.net` (Kevin Lahey) suggests that this is out-of-date, and 2312 (WiFi 802.11), 9180 (common ATM), and 9000 (jumbo Ethernet) should be added. For some polemic discussion, see <http://www.psc.edu/~mathis/MTU/>.

RFC1191 says explicitly “We do not expect that the values in the table [...] are going to be valid forever. The values given here are an implementation suggestion, NOT a specification or requirement. Implementors should use up-to-date references to pick a set of plateaus [...]”. BSD is therefore not compliant here.

Linux adds 576, 216, 128 and drops 1006. 576 is used in X.25 networks, and the source says 216 and 128 are needed for AMPRnet AX.25 paths. 1006 is used for SLIP, and was used on the ARPANET. Linux does not include the modern MTUs listed above.

12.14 Reassembly (TCP only)

TCP segments may arrive out-of-order, leaving holes in the data stream. They may also overlap, due to retransmission, confusion, or deliberate effort by an unusual TCP implementation. The TCP reassembly algorithm is responsible for retrieving the data stream from the segments that arrive (note this is not to be confused with IP fragmentation reassembly, which is beneath the scope of this specification).

There are various ways of resolving overlaps; in this specification we are completely nondeterministic, and allow any legal reassembly.

12.14.1 Summary

| | |
|------------------------|---------------------------------|
| <i>tcp_reass</i> | perform TCP segment reassembly |
| <i>tcp_reass_prune</i> | drop prefix of reassembly queue |

12.14.2 Rules

– **perform TCP segment reassembly :**

tcp_reass seq(*rseqq* : tcpReassSegment list) =

```

let myrel = {(i, c) | ∃rseq.
    rseq ∈ rseqq ∧
    i ≥ rseq.seq ∧
    i < rseq.seq + length rseq.data +
      (if rseq.spliced_urp ≠ * then 1 else 0) ∧
    (case rseq.spliced_urp of
      ↑(n) →
        (if i > n then
          c = ↑(EL(num(i − rseq.seq − 1))(rseq.data))
        else if i = n then
          c = *
        else
          c = ↑(EL(num(i − rseq.seq))(rseq.data)) ||
        * →
          c = ↑(EL(num(i − rseq.seq))(rseq.data))} in
    {(cs', len, FIN) | ∃cs.cs' = CONCAT_OPTIONAL cs ∧
      (∀n : num.n < length cs ⇒ (seq + n, EL n cs) ∈ myrel) ∧
      (¬∃c.(seq + length cs, c) ∈ myrel) ∧
      (len = length cs) ∧
      (FIN = ∃rseq.rseq ∈ rseqq ∧
        rseq.seq + length rseq.data +
          (if rseq.spliced_urp ≠ * then 1 else 0) =
          seq + length cs ∧
          rseq.FIN)}
```

(* NB: the FIN may come from a 0-length segment, or from a different segment from that which the last character came but logically is always at the end of *cs*'s. *)

Description Returns the set of maximal-length strings starting at *seq* that can be constructed by taking bytes from the segments in *rseqq*, accounting for any spliced (out-of-line) urgent data.

```

– drop prefix of reassembly queue :
tcp_reass_prune seq(rseqq : tcpReassSegment list) =
filter(λrseq.rseq.seq + length rseq.data + (if rseq.spliced_urp ≠ * then 1 else 0) +
      (if rseq.FIN then 1 else 0) > seq)rseqq

```

Description Prune away every segment ending before the specified *seq*, accounting for any spliced (out-of-line) urgent data.

12.15 The initial TCP control block (TCP only)

The initial state of the TCP control block.

12.15.1 Summary

initial_cb

12.15.2 Rules

```

– :
initial_cb =
⟦ t_seqq := [];
  tt_rexmt := *;
  tt_keep := *;
  tt_2msl := *;
  tt_delack := *;
  tt_conn_est := *;
  tt_fin_wait_2 := *;
  tf_needfin := F;
  tf_shouldacknow := F;
  snd_una := tcp_seq_local 0w;
  snd_max := tcp_seq_local 0w;
  snd_nxt := tcp_seq_local 0w;
  snd_wl1 := tcp_seq_foreign 0w;
  snd_wl2 := tcp_seq_local 0w;
  iss := tcp_seq_local 0w;
  snd_wnd := 0;
  snd_cwnd := TCP_MAXWIN ≪ TCP_MAXWINSIZE;
  snd_ssthresh := TCP_MAXWIN ≪ TCP_MAXWINSIZE;
  rcv_wnd := 0;
  tf_rxwin0sent := F;
  rcv_nxt := tcp_seq_foreign 0w;
  rcv_up := tcp_seq_foreign 0w;
  irs := tcp_seq_foreign 0w;
  rcv_adv := tcp_seq_foreign 0w;
  snd_recover := tcp_seq_local 0w;
  t_maxseg := MSSDFLT;
  t_advmss := *;
  t_rttseg := *;
  t_rttinf :=
⟧

```

```

    t_rttupdated := 0;
    tf_srtt_valid := F;
    t_srtt := TCPTV_RTOBASE;
    t_rttvar := TCPTV_RTTVARBASE;
    t_rttmin := TCPTV_MIN;
    t_lastrtt := 0;
    t_lastshift := 0;
    t_wassyn := F (* if t_lastshift=0, this doesn't make a difference *)
  };
  t_dupacks := 0;
  t_idletime := stopwatch_zero;
  t_softerror := *;
  snd_scale := 0;
  rcv_scale := 0;
  request_r_scale := *; (* this like many other things is overwritten with the chosen value later - cf tcp_newtcpb() *)
  tf_doing_ws := F;
  ts_recent := TIMEWINDOWCLOSED;
  tf_req_tstamp := F; (* cf tcp_newtcpb() *)
  tf_doing_tstamp := F;
  last_ack_sent := tcp_seq_foreign 0w;
  bsd_cantconnect := F;
  snd_cwnd_prev := 0;
  snd_ssthresh_prev := 0;
  t_badrtwin := TIMEWINDOWCLOSED
  (* Note: everything should be listed here, leaving nothing as ARB. *)
  (* Many are always overwritten, however. *)
}

```

Chapter 13

Relational monad

The relational ‘monad’ is used to describe stateful computation in a convenient and compositional way.

13.1 Relational monad (TCP only)

The implementation TCP input and output routines are imperative C code, with mutations of state variables and calls to various other routines, some of which send messages or have other observable effects. These are intertwined in a complex control flow. In the specification we have attempted, as much as possible, to adopt purely functional or relational styles. To deal with the observable side effects in the middle of (e.g.) `tcp_output`, however, we have had to identify some intermediate states. We introduce a relational monadic style to do so, using higher-order functions to hide the plumbing of state variables. The nondeterminism of our model adds another layer of complexity; instead of the usual functional monads, we use relational monads.

An operation on the current state is modelled by a relation on the current and resulting states. A number of primitive operations are defined; these operations are then chained together by a binding combinator, which takes two relations and yields their composition. In this way arbitrarily complex operations on state may be defined in a modular manner, and the referential transparency of the logic is maintained.

In the present application, the current state is a pair (`sock` : `socket`, `bndlm` : `bandlim_state`) of the current socket and the state of the host’s band limiter. The resulting state is a quadruple ((`sock'` : `socket`, `bndlm'` : `bandlim_state`, `outsegs'` : 'msglist'), `continue'` : `bool`) of the final socket, band-limiter state, a list of segments to be output, and a flag. This flag models aborting: if it is set, operations should be chained together normally; if it is cleared, subsequent operations should *not* be performed, and instead the resulting state should be the final state of the entire composite operation of which this is a part.

The binding combinator is `andThen`. Primitive operators include `cont`, which does nothing and continues, and `stop`, which does nothing and stops. Several other operations are defined to manipulate the state – the monadic glue is intended to abstract away from the implementation of that state as a pair of tuples.

It should be a theorem that `andThen` is `assoc`, that `cont` is `unit` and `stop` is `zero`, and so on.

Note that `outsegs`, the list of messages, is actually a list of arbitrary type; this enables us to lift the glue to the type `msg#bool` in `deliver_in_3`, where we need the flag to deal with queuing failure.

As throughout this specification, beware that the nondeterminism of, e.g., `chooseM` is modelled by an existential, and is thus “angelic” in some sense. This may or may not be what you expect.

13.1.1 Summary

| | |
|------------------------------|---|
| <code>andThen</code> | normal sequencing |
| <code>cont</code> | do nothing, and continue (unit for <code>andThen</code>) |
| <code>stop</code> | do nothing, and stop (zero for <code>andThen</code>) |
| <code>assert</code> | assert truth of condition, and continue |
| <code>assert_failure</code> | assertion violated; fail noisily |
| <code>chooseM</code> | choose a value from a set, nondeterministically |
| <code>get_sock</code> | get current socket |
| <code>get_tcp_sock</code> | assert current socket is TCP, and get its protocol data |
| <code>get_cb</code> | assert current socket is TCP, and get its control block |
| <code>modify_sock</code> | apply function to current socket |
| <code>modify_tcp_sock</code> | apply function to current socket |

| | |
|-----------------------|--|
| <i>modify_cb</i> | assert current socket is TCP, and apply function to its control block |
| <i>emit_segs</i> | append segments to current output list |
| <i>emit_segs_pred</i> | append segments specified by a predicate (nondeterministic) |
| <i>mliftc</i> | lift a monadic operation not involving <i>continue</i> or <i>bndlm</i> |
| <i>mliftc_bndlm</i> | lift a monadic operation not involving <i>continue</i> |

13.1.2 Rules

– **normal sequencing :**

op1 andThen op2 =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$\exists \text{sock}_1 \text{ bndlm}_1 \text{ outsegs}_1 \text{ continue}_1 \text{ sock}_2 \text{ bndlm}_2 \text{ outsegs}_2 \text{ continue}_2.$

$\text{op1}(\text{sock}, \text{bndlm})((\text{sock}_1, \text{bndlm}_1, \text{outsegs}_1), \text{continue}_1) \wedge$

if *continue₁* **then**

$\text{op2}(\text{sock}_1, \text{bndlm}_1)((\text{sock}_2, \text{bndlm}_2, \text{outsegs}_2), \text{continue}_2) \wedge$

$(\text{sock}' = \text{sock}_2 \wedge \text{bndlm}' = \text{bndlm}_2 \wedge \text{outsegs}' = \text{outsegs}_1 @ \text{outsegs}_2 \wedge \text{continue}' = \text{continue}_2)$

else

$(\text{sock}' = \text{sock}_1 \wedge \text{bndlm}' = \text{bndlm}_1 \wedge \text{outsegs}' = \text{outsegs}_1 \wedge \text{continue}' = \mathbf{F})$

– **do nothing, and continue (unit for andThen) :**

cont =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\text{sock}' = \text{sock} \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{T})$

– **do nothing, and stop (zero for andThen) :**

stop =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\text{sock}' = \text{sock} \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{F})$

– **assert truth of condition, and continue :**

assert p =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(p \wedge \text{sock}' = \text{sock} \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{T})$

– **assertion violated; fail noisily :**

assert_failure s =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

ASSERTION_FAILURE s

– **choose a value from a set, nondeterministically :**

chooseM s f =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

choose *x* :: *s.f x(sock, bndlm)((sock', bndlm', outsegs'), continue')*

– **get current socket :**

get_sock f =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

f sock(sock, bndlm)((sock', bndlm', outsegs'), continue')

– **assert current socket is TCP, and get its protocol data :**

get_tcp_sock f =

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$\exists \text{tcp_sock}.$

sock.pr = *TCP_PROTO(tcp_sock)* \wedge

f tcp_sock(sock, bndlm)((sock', bndlm', outsegs'), continue')

– **assert current socket is TCP, and get its control block :**

get_cb $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$
 $\exists \text{tcp_sock}.$

$\text{sock.pr} = \text{TCP_PROTO}(\text{tcp_sock}) \wedge$

$f \text{ tcp_sock.cb}(\text{sock}, \text{bndlm})((\text{sock}', \text{bndlm}', \text{outsegs}'), \text{continue}')$

– **apply function to current socket :**

modify_sock $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\text{sock}' = f \text{ sock} \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{T})$

– **apply function to current socket :**

modify_tcp_sock $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\exists \text{tcp_sock}.$

$\text{sock.pr} = \text{TCP_PROTO}(\text{tcp_sock}) \wedge$

$\text{sock}' = \text{sock} \llbracket \text{pr} := \text{TCP_PROTO}(f \text{ tcp_sock}) \rrbracket \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{T})$

– **assert current socket is TCP, and apply function to its control block :**

modify_cb $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$\exists \text{tcp_sock}.$

$\text{sock.pr} = \text{TCP_PROTO}(\text{tcp_sock}) \wedge$

$(\text{sock}' = \text{sock} \llbracket \text{pr} := \text{TCP_PROTO}(\text{tcp_sock} \llbracket \text{cb} := (f \text{ tcp_sock.cb}) \rrbracket) \rrbracket) \wedge$

$\text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = [] \wedge \text{continue}' = \mathbf{T})$

– **append segments to current output list :**

emit_segs $\text{segs} =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\text{sock}' = \text{sock} \wedge \text{bndlm}' = \text{bndlm} \wedge \text{outsegs}' = \text{segs} \wedge \text{continue}' = \mathbf{T})$

– **append segments specified by a predicate (nondeterministic) :**

emit_segs_pred $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(\text{sock}' = \text{sock} \wedge f \text{ bndlm} \text{ bndlm}' \text{ outsegs}' \wedge \text{continue}' = \mathbf{T})$

– **lift a monadic operation not involving continue or bndlm :**

mliftc $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(f \text{ sock}(\text{sock}', \text{outsegs}') \wedge \text{bndlm}' = \text{bndlm} \wedge \text{continue}' = \mathbf{T})$

– **lift a monadic operation not involving continue :**

mliftc_bndlm $f =$

$\lambda(\text{sock} : \text{socket}, \text{bndlm} : \text{bandlim_state})((\text{sock}' : \text{socket}, \text{bndlm}' : \text{bandlim_state}, \text{outsegs}' : 'msg \text{ list}), \text{continue}' : \text{bool}).$

$(f(\text{sock}, \text{bndlm})(\text{sock}', \text{bndlm}', \text{outsegs}') \wedge \text{continue}' = \mathbf{T})$

Chapter 14

Auxiliary functions for TCP segment creation and drop

We gather here all the general TCP segment generation and processing functions that are used in the host LTS.

14.1 SYN and RST Segment Creation (TCP only)

Generating various simple segments (none of which contain any user data).

14.1.1 Summary

| | |
|----------------------------------|---|
| <i>make_syn_segment</i> | Make a SYN segment for emission by <i>connect_1</i> etc |
| <i>make_syn_ack_segment</i> | Make a SYN,ACK segment for emission by <i>deliver_in_1</i> , <i>deliver_in_2</i> , etc. |
| <i>make_ack_segment</i> | Make a plain boring ACK segment in response to a SYN,ACK segment |
| <i>bsd_make_phantom_segment</i> | Make phantom (no flags) segment for BSD LISTEN bug |
| <i>make_rst_segment_from_cb</i> | Make a RST segment asynchronously, from socket information only |
| <i>make_rst_segment_from_seg</i> | Make a RST segment synchronously, in response to an incoming segment |

14.1.2 Rules

– **Make a SYN segment for emission by *connect_1* etc :**

```
make_syn_segment cb(i1, i2, p1, p2)ts_val seg' =  
(choose urp_any :: UNIV.  
choose ack_any :: UNIV.
```

(* Determine window size; fail if out of range *)

```
let win = n2w cb.rcv_wnd in  
w2n win = cb.rcv_wnd ∧
```

(* Choose a window scaling; fail if out of range *)

(* Note there may be a better place for this assertion. *)

```
let ws = option_map CHR cb.request_r_scale in  
(is_some cb.request_r_scale ⇒ ord(the ws) = the cb.request_r_scale) ∧  
(case ws of * → T || ↑ n → ord n ≤ TCP_MAXWINSIZE) ∧
```

```

(* Determine maximum segment size; fail if out of range *)
(* Put the MSS we initially advertise into t_advmss *)
let mss = (case cb.t_advmss of
    * → *
    || ↑ v → ↑(n2w v)) in
(case cb.t_advmss of
    * → T
    || ↑ v → v = w2n(the mss)) ∧

(* Do timestamping? *)
let ts = do_tcp_options cb.tf_req_tstamp cb.ts_recent ts_val in

seg' = [ is1 := ↑ i1;
        is2 := ↑ i2;
        ps1 := ↑ p1;
        ps2 := ↑ p2;
        seq := cb.iss;
        ack := ack_any;
        URG := F;
        ACK := F;
        PSH := F;
        RST := F;
        SYN := T;
        FIN := F;
        win := win;
        ws := ws;
        urp := urp_any;
        mss := mss;
        ts := ts;
        data := []
      ]
)

```

– **Make a SYN,ACK segment for emission by deliver_in_1, deliver_in_2, etc.:**

```
make_syn_ack_segment cb(i1, i2, p1, p2)ts_val' seg' =
```

```
choose urp_any :: UNIV.
```

```
(* Determine window size; fail if out of range *)
```

```
(* We don't scale yet ( $\gg$  rcv_scale'). RFC1323 says: segments with SYN are not scaled, and BSD agrees. Even though we know what scaling the other end wants to use, and we know whether we are doing scaling, we can't use it until we reach the ESTABLISHED state. *)
```

```
let win = n2w cb.rcv_wnd in (* rcv_window - length data' *)
```

```
w2n win = cb.rcv_wnd ∧
```

```
(* If doing window scaling, set it; fail if out of range *)
```

```
let ws = if cb.tf_doing_ws then ↑(CHR cb.rcv_scale) else * in
```

```
(cb.tf_doing_ws  $\implies$  ord(the ws) = cb.rcv_scale) ∧
```

```
(* Determine maximum segment size; fail if out of range *)
```

```
(* Put the MSS we initially advertise into t_advmss *)
```

```
let mss = (case cb.t_advmss of
```

```
    * → *
```

```
    || ↑ v → ↑(n2w v)) in
```

```
(case cb.t_advmss of
```

```
    * → T
```

```
    || ↑ v → v = w2n(the mss)) ∧
```

(* Set timestamping option? *)

let *ts* = do_tcp_options *cb.tf_doing_tstmp* *cb.ts_recent* *ts_val'* **in**

```

seg' = [ is1 := ↑ i1;
         is2 := ↑ i2;
         ps1 := ↑ p1;
         ps2 := ↑ p2;
         seq := cb.iss;
         ack := cb.rcv_nxt;
         URG := F;
         ACK := T;
         PSH := F; (* see below *)
         RST := F;
         SYN := T;
         FIN := F; (* Note: we are not modelling T/TCP *)
         win := win;
         ws := ws;
         urp := urp_any;
         mss := mss;
         ts := ts;
         data := [] (* see below *)
       ]

```

(* No *data* can be send here using the BSD sockets API, although TCP notionally allows it. Accordingly, the *PSH* flag is never set (under BSD, *PSH* is only set if we're sending a non-zero amount of data (and emptying the send buffer); see `tcp_output.c:626`). *)

– **Make a plain boring ACK segment in response to a SYN,ACK segment :**

make_ack_segment *cb* *FIN*(*i*₁, *i*₂, *p*₁, *p*₂)*ts_val'* *seg'* =

((* SB thinks these should be unconstrained. *)

choose *urp_garbage* :: *UNIV*.

(* Determine window size; fail if out of range *)

(* Connection is now established so any scaling should be taken into account *)

(* Note it might be appropriate to clip the value to be in range rather than failing if out of range. *)

let *win* = **n2w**(*cb.rcv_wnd* >>> *cb.rcv_scale*) **in**

w2n *win* = *cb.rcv_wnd* >>> *cb.rcv_scale* ∧

(* Set timestamping option? *)

let *ts* = do_tcp_options *cb.tf_doing_tstmp* *cb.ts_recent* *ts_val'* **in**

```

seg' = [ is1 := ↑ i1;
         is2 := ↑ i2;
         ps1 := ↑ p1;
         ps2 := ↑ p2;
         seq := if FIN then cb.snd_una else cb.snd_nxt;
         ack := cb.rcv_nxt;
         URG := F;
         ACK := T;
         PSH := F; (* see comment for make_syn_ack_segment *)
         RST := F;
         SYN := F;
         FIN := FIN;
         win := win;
         ws := *;
         urp := urp_garbage;
       ]

```

```

    mss := *;
    ts := ts;
    data := [] (* Note that if there is data in sndq then it should always appear in a separate segment after the
    connection establishment handshake, but this needs to be verified. *)
  ))

```

– **Make phantom (no flags) segment for BSD LISTEN bug :**

(* If a socket is changed to the LISTEN state, the rexmt timer may still be running. If it fires, phantom segments are emitted. *)

```

bsd_make_phantom_segment cb(i1, i2, p1, p2)ts_val' cantsndmore seg' =
  (choose urp_garbage :: UNIV.

```

(* Determine window size; fail if out of range *)

(* Connection is now established so any scaling should be taken into account *)

(* Note it might be appropriate to clip the value to be in range rather than failing if out of range. *)

```

let win = n2w(cb.rcv_wnd >> cb.rcv_scale) in
w2n win = cb.rcv_wnd >> cb.rcv_scale ∧

```

```

let FIN = (cantsndmore ∧ cb.snd_una < (cb.snd_max - 1)) in

```

(* Set timestamping option? *)

```

let ts = do_tcp_options cb.tf_doing_tstamp cb.ts_recent ts_val' in

```

```

seg' = [ is1 := ↑ i1;
        is2 := ↑ i2;
        ps1 := ↑ p1;
        ps2 := ↑ p2;
        seq := if FIN then cb.snd_una else cb.snd_max; (* no flags, no data, and no persist timer so use snd_max *)
        ack := cb.rcv_next; (* yes, really, even though ¬ACK *)
        URG := F;
        ACK := F;
        PSH := F;
        RST := F;
        SYN := F;
        FIN := FIN;
        win := win;
        ws := *;
        urp := urp_garbage;
        mss := *;
        ts := ts;
        data := [] (* sndq always empty in this situation *)
      ]
  ))

```

– **Make a RST segment asynchronously, from socket information only :**

```

make_rst_segment_from_cb cb(i1, i2, p1, p2)seg' =

```

(* Deliberately unconstrained *)

```

choose urp_garbage :: UNIV.
choose URG_garbage :: UNIV.
choose PSH_garbage :: UNIV.
choose win_garbage :: UNIV.
choose data_garbage :: UNIV.
choose FIN_garbage :: UNIV.

```

(* Note that BSD is perfectly capable of putting data in a RST segment; try filling the buffer and then doing a force close: the result is a segment with RST+PSH+data+win advertisement. Presumably URG is also possible. This is *not* the same as the RFC-suggested data carried by a RST; that would be an error message, this is just data from the buffer! *)

```

seg' = ( [ is1 := ↑ i1;
        ps1 := ↑ p1;
        is2 := ↑ i2;
        ps2 := ↑ p2;
        seq := cb.snd_next; (* from RFC793p62 *)
        ack := cb.rcv_next; (* seems the right thing to do *)
        URG := URG_garbage; (* expect: F *)
        ACK := T; (* from TCPv1p248 *)
        PSH := PSH_garbage; (* expect: F *)
        RST := T;
        SYN := F;
        FIN := FIN_garbage; (* expect: F *)
        win := win_garbage; (* expect: 0w *)
        ws := *;
        urp := urp_garbage; (* expect: 0w *)
        mss := *;
        ts := *; (* RFC1323 S4.2 recommends no TS on RST, and BSD follows this *)
        data := data_garbage (* expect: [] *)
      ]
    )

```

– **Make a RST segment synchronously, in response to an incoming segment :**

```

make_rst_segment_from_seg seg seg' =
(seg.RST = F ∧ (* Sanity check: never RST a RST *))

```

```

(∃ ack'.
(* Deliberately unconstrained *)
choose urp_garbage :: UNIV.
choose URG_garbage :: UNIV.
choose PSH_garbage :: UNIV.
choose win_garbage :: UNIV.
choose data_garbage :: UNIV.
choose FIN_garbage :: UNIV.

```

(* RFC795 S3.4: only ack segments that don't contain an ACK. SB believes this is equivalent to: only send a RST+ACK segment in response to a bad SYN segment *)

```

let ACK' = ¬seg.ACK in

```

(* Sequence number is zero for RST+ACK segments, otherwise it is the next sequence number expected *)

```

let seq' = if seg.ACK then tcp_seq_flip_sense seg.ack
           else tcp_seq_local 0w in

```

```

(if ACK' then

```

(* RFC794 S3.4: for RST+ACK segments the ack value must be valid *)

```

  ack' = tcp_seq_flip_sense seg.seq + length seg.data + (if seg.SYN then 1 else 0)

```

```

else

```

(* otherwise it can be arbitrary, although it possibly should be zero *)

```

  ack' ∈ {n | T}

```

```

) ∧

```

```

seg' = ( [ is1 := seg.is2;
        ps1 := seg.ps2;
        is2 := seg.is1;
        ps2 := seg.ps1;
        seq := seq';

```

```

    ack := ack';
    URG := URG_garbage; (* expect: F *)
    ACK := ACK';
    PSH := PSH_garbage; (* expect: F *)
    RST := T;
    SYN := F;
    FIN := FIN_garbage; (* expect: F *)
    win := win_garbage; (* expect: 0w *)
    ws := *;
    urp := urp_garbage; (* expect: 0w *)
    mss := *;
    ts := *; (* RFC1323 S4.2 recommends no TS on RST, and BSD follows this *)
    data := data_garbage (* expect: [] *)
  )
))

```

14.2 General Segment Creation (TCP only)

The TCP output routines. These, together with the input routines in *deliver_in_3*, form the heart of TCP.

14.2.1 Summary

| | |
|----------------------------|--|
| <i>tcp_output_required</i> | determine whether TCP output is required |
| <i>tcp_output_really</i> | do TCP output |
| <i>tcp_output_perhaps</i> | combination of <i>tcp_output_required</i> and <i>tcp_output_really</i> |

14.2.2 Rules

– determine whether TCP output is required :

```
tcp_output_required arch ifds0 sock =
```

```
let tcp_sock = tcp_sock_of sock in
```

```
let cb = tcp_sock.cb in
```

```
(* Note this does not deal with TF_LASTIDLE and PRU_MORETOCOME *)
```

```
let snd_cwnd' =
```

```
if ¬(cb.snd_max = cb.snd_una ∧
```

```
stopwatch_val_of cb.t_idletime ≥ computed_rxtcur cb.t_rttinf arch)
```

```
then (* inverted so this clause is tried first *)
```

```
cb.snd_cwnd
```

```
else
```

```
(* The connection is idle and has been for ≥ 1 RTO *)
```

```
(* Reduce snd_cwnd to commence slow start *)
```

```
cb.t_maxseg * (if is_localnet ifds0(the sock.is2) then SS_FLTSZ_LOCAL else SS_FLTSZ) in
```

```
(* Calculate the amount of unused send window *)
```

```
let win = min cb.snd_wnd snd_cwnd' in
```

```
let snd_wnd_unused = int_of_num win - (cb.snd_nxt - cb.snd_una) in
```

```
(* Is it possible that a FIN may need to be sent? *)
```

```
let fin_required = (sock.cantsndmore ∧ tcp_sock.st ∉ {FIN_WAIT_2; TIME_WAIT}) in
```

```
(* Under BSD, we may need to send a FIN in state SYN_SENT or SYN_RECEIVED, so we may effectively still have a SYN on the send queue. *)
```



```

let syn_not_acked = (bsd_arch arch ∧ tcp_sock.st ∈ {SYN_SENT; SYN_RECEIVED}) in

  (* Is there data or a FIN to transmit? *)
let last_sndq_data_seq = cb.snd_una + length tcp_sock.sndq in
let last_sndq_data_and_fin_seq = last_sndq_data_seq + (if fin_required then 1 else 0)
                                          + (if syn_not_acked then 1 else 0) in
let have_data_to_send = cb.snd_nxt < last_sndq_data_seq in
let have_data_or_fin_to_send = cb.snd_nxt < last_sndq_data_and_fin_seq in

  (* The amount by which the right edge of the advertised window could be moved *)
let window_update_delta = (int_min(int_of_num(TCP_MAXWIN ≪ cb.rcv_scale))
                          (int_of_num(sock.sf.n(SO_RCVBUF)) - int_of_num(length
                          tcp_sock.rcvq))) -
  (cb.rcv_adv - cb.rcv_nxt) in

  (* Send a window update? This occurs when (a) the advertised window can be increased by at least two maximum segment sizes, or (b) the advertised window can be increased by at least half the receive buffer size. See tcp_output.c:322ff. *)
let need_to_send_a_window_update = (window_update_delta ≥ int_of_num(2 * cb.t_maxseg) ∨
  2 * window_update_delta ≥ int_of_num(sock.sf.n(SO_RCVBUF)))
in

  (* Note that silly window avoidance and max_sndwnd need to be dealt with here; see tcp_output.c:309 *)

  (* Can a segment be transmitted? *)
let do_output = (
  (* Data to send and the send window has some space, or a FIN can be sent *)
  (have_data_or_fin_to_send ∧
  (have_data_to_send ⇒ snd_wnd_unused > 0)) ∨ (* don't need space if only sending FIN *)

  (* Can send a window update *)
  need_to_send_a_window_update ∨

  (* There is outstanding urgent data to be transmitted *)
  is_some tcp_sock.sndurp ∨

  (* An ACK should be sent immediately (e.g. in reply to a window probe) *)
  cb.tf_shouldacknow
  ) in

let persist_fun =
let cant_send = (¬do_output ∧ tcp_sock.sndq ≠ [] ∧ mode_of cb.tt_rexmt = *) in
let window_shrunk = (win = 0 ∧ snd_wnd_unused < 0 ∧ (* win = 0 if in SYN_SENT, but still may send FIN *)
  (bsd_arch arch ⇒ tcp_sock.st ≠ SYN_SENT)) in

if cant_send then (* takes priority over window_shrunk; note this needs to be checked *)
  (* Can not transmit a segment despite a non-empty send queue and no running persist or retransmit timer. Must be the case that the receiver's advertised window is now zero, so start the persist timer. Normal: tcp_output.c:378ff *)
  ↑λcb.cb { tt_rexmt := start_tt_persist 0 cb.t_rttinf arch }
else if window_shrunk then
  (* The receiver's advertised window is zero and the receiver has retracted window space that it had previously advertised. Reset snd_nxt to snd_una because the data from snd_una to snd_nxt has likely not been buffered by the receiver and should be retransmitted. Bizarrely (on FreeBSD 4.6-RELEASE), if the persist timer is running reset its shift value *)
  (* Window shrunk: |tcp_output.c:250ff| *)
  ↑λcb.
  cb { tt_rexmt := case cb.tt_rexmt of
    ↑(((PERSIST, shift))d) → ↑(((PERSIST, 0))d)
    || _593 → start_tt_persist 0 cb.t_rttinf arch;

```

```

        snd_nxt := cb.snd_una)
else
  (* Otherwise, leave the persist timer alone *)
  *
in
(do_output, persist_fun)

```

Description

This function determines if it is currently necessary to emit a segment. It is not quite a predicate, because in certain circumstances the operation of testing may start or reset the persist timer, and alter *snd_nxt*. Thus it returns a pair of a flag *do_output* (with the obvious meaning), and an optional mutator function *persist_fun* which, if present, performs the required updates on the TCP control block.

– do TCP output :

```
tcp_output_really arch window_probe ts_val' ifds0 sock(sock', outsegs') =
```

```
let tcp_sock = tcp_sock_of sock in
```

```
let cb = tcp_sock.cb in
```

```
  (* Assert that the socket is fully bound and connected *)
```

```
sock.is1 ≠ * ∧
```

```
sock.is2 ≠ * ∧
```

```
sock.ps1 ≠ * ∧
```

```
sock.ps2 ≠ * ∧
```

```
  (* Note this does not deal with TF_LASTIDLE and PRU_MORETOCOME *)
```

```
let snd_cwnd' =
```

```
if ¬(cb.snd_max = cb.snd_una ∧
```

```
    stopwatch_val_of cb.t_idletime ≥ computed_rxtcur cb.t_rttinf arch)
```

```
  then (* inverted so this clause is tried first *)
```

```
    cb.snd_cwnd
```

```
  else
```

```
    (* The connection is idle and has been for ≥ 1RTO *)
```

```
    (* Reduce snd_cwnd to commence slow start *)
```

```
    cb.t_maxseg * (if is_localnet ifds0(the sock.is2) then SS_FLTSZ_LOCAL else SS_FLTSZ) in
```

```
  (* Calculate the amount of unused send window *)
```

```
let win0 = min cb.snd_wnd snd_cwnd' in
```

```
let win = (if window_probe ∧ win0 = 0 then 1 else win0) in
```

```
let (snd_wnd_unused : int) = int_of_num win - (cb.snd_nxt - cb.snd_una) in
```

```
  (* Is it possible that a FIN may need to be transmitted? *)
```

```
let fin_required = (sock.cantsndmore ∧ tcp_sock.st ∉ {FIN_WAIT_2; TIME_WAIT}) in
```

```
  (* Calculate the sequence number after the last data byte in the send queue *)
```

```
let last_sndq_data_seq = cb.snd_una + length tcp_sock.sndq in
```

```
  (* The data to send in this segment (if any) *)
```

```
let data' = DROP(num(cb.snd_nxt - cb.snd_una))tcp_sock.sndq in
```

```
let data_to_send = TAKE(min(clip_int_to_num snd_wnd_unused)cb.t_maxseg)data' in
```

```
  (* Should FIN be set in this segment? *)
```

```
let FIN = (fin_required ∧ cb.snd_nxt + length data_to_send ≥ last_sndq_data_seq) in
```

```
  (* Should ACK be set in this segment? Under BSD, it is not set if the socket is in SYN_SENT and emitting a FIN segment due to shutdown() having been called. *)
```

```
let ACK = if (bsd_arch arch ∧ FIN ∧ tcp_sock.st = SYN_SENT) then F else T in
```

(* If this socket has previously sent a *FIN* which has not yet been acked, and *snd_nxt* is past the *FIN*'s sequence number, then *snd_nxt* should be set to the sequence number of the *FIN* flag, i.e. a retransmission. Check that *snd_una* \neq *iss* as in this case no data has yet been sent over the socket *)

```
let snd_nxt' = if FIN  $\wedge$  (cb.snd_nxt + length data_to_send = last_sndq_data_seq + 1  $\wedge$ 
    cb.snd_una  $\neq$  cb.iss  $\vee$  num(cb.snd_nxt - cb.iss) = 2) then
    cb.snd_nxt - 1
else
    cb.snd_nxt in
```

(* The BSD way: set *PSH* whenever sending the last byte of data in the send queue *)

```
let PSH = (data_to_send  $\neq$  []  $\wedge$  cb.snd_nxt + length data_to_send = last_sndq_data_seq) in
```

(* If sending urgent data, set the *URG* and *urp* fields appropriately *)

```
let (URG, urp) = (case tcp_sock.sndurp of
    *  $\rightarrow$  (F, 0) || (* No urgent data; don't set *)
     $\uparrow$  sndurpn  $\rightarrow$  let urpn = (cb.snd_una + sndurpn) - cb.snd_nxt + 1 in
        (* points one byte *past* the urgent byte *)
        if urpn < 1 then
            (F, 0) (* Urgent data out of range; don't set *)
        else if urpn < 65536 then
            (T, num urpn) (* Urgent data in range; set *)
        else
            (* Urgent data in the very distant future; set *)
            (* Steven's suggestion; not sure if followed *)
            (T, 65535) in
```

(* Calculate size of the receive window (based upon available buffer space) *)

```
let rcv_wnd'' = calculate_bsd_rcv_wnd sock.sf tcp_sock in
let rcv_wnd' = max(num(cb.rcv_adv - cb.rcv_nxt))(min(TCP_MAXWIN  $\ll$  cb.rcv_scale)
    (if rcv_wnd'' < sock.sf.n(SO_RCVBUF) div 4  $\wedge$  rcv_wnd'' < cb.t_maxseg
        then 0 (* Silly window avoidance: shouldn't advertise a tiny window *)
        else rcv_wnd'')) in
```

(* Possibly set the segment's timestamp option. Under BSD, we may need to send a *FIN* segment from SYN_SENT, if the user called *shutdown*(), in which case the timestamp option hasn't yet been negotiated, so we used *tf_req_tstamp* rather than *tf_doing_tstamp*. *)

```
let want_tstamp = if (bsd_arch arch  $\wedge$  tcp_sock.st = SYN_SENT) then cb.tf_req_tstamp
    else cb.tf_doing_tstamp in
```

```
let ts = do_tcp_options want_tstamp cb.ts_recent ts_val' in
```

(* Advertise an appropriately scaled receive window *)

(* Assert the advertised window is within a sensible range *)

```
let win = n2w(rcv_wnd'  $\gg$  cb.rcv_scale) in
w2n win = rcv_wnd'  $\gg$  cb.rcv_scale  $\wedge$ 
```

(* Assert the urgent pointer is within a sensible range *)

```
let urp_ = n2w urp in
w2n urp_ = urp  $\wedge$ 
```

```
let seg =  $\langle$  is1 := sock.is1;
    is2 := sock.is2;
    ps1 := sock.ps1;
    ps2 := sock.ps2;
    seq := snd_nxt';
    ack := cb.rcv_nxt;
    URG := URG;
    ACK := ACK;
    PSH := PSH;
```

```

    RST := F;
    SYN := F;
    FIN := FIN;
    win := win;
    ws := *;
    urp := urp_;
    mss := *;
    ts := ts;
    data := data_to_send
  } in

```

(* If emitting a *FIN* for the first time then change TCP state *)

```

let st' = if FIN then
  case tcp_sock.st of
    SYN_SENT → tcp_sock.st || (* can't move yet - wait until connection established (see
                               deliver_in_2/deliver_in_3) *)
    SYN_RECEIVED → tcp_sock.st || (* can't move yet - wait until connection established (see
                                   deliver_in_2/deliver_in_3) *)
    ESTABLISHED → FIN_WAIT_1 ||
    CLOSE_WAIT → LAST_ACK ||
    FIN_WAIT_1 → tcp_sock.st || (* FIN retransmission *)
    FIN_WAIT_2 → tcp_sock.st || (* can't happen *)
    CLOSING → tcp_sock.st || (* FIN retransmission *)
    LAST_ACK → tcp_sock.st || (* FIN retransmission *)
    TIME_WAIT → tcp_sock.st (* can't happen *)
  else
    tcp_sock.st in

```

(* Updated values to store in the control block after the segment is output *)

```

let snd_nxt'' = snd_nxt' + length data_to_send + (if FIN then 1 else 0) in
let snd_max' = max cb.snd_max snd_nxt'' in

```

(* Following a `tcp_output` code walkthrough by SB: *)

```

let tt_rexmt' = if (mode_of cb.tt_rexmt = * ∨
  (mode_of cb.tt_rexmt = ↑(PERSIST) ∧ ¬window_probe)) ∧
  snd_nxt'' > cb.snd_una then
  (* If the retransmit timer is not running, or the persist timer is running and this segment isn't
  a window probe, and this segment contains data or a FIN that occurs past snd_una (i.e. new
  data), then start the retransmit timer. Note: if the persist timer is running it will be implicitly
  stopped *)
  start_tt_rexmt arch 0 F cb.t_rttinf
else if (window_probe ∨ (is_some tcp_sock.sndurp)) ∧ win_0 ≠ 0 ∧
  mode_of cb.tt_rexmt = ↑(PERSIST) then
  (* If the segment is a window probe or urgent data is being sent, and in either case the send
  window is not closed, stop any running persist timer. Note: if window_probe is T then a persist
  timer will always be running but this isn't necessarily true when urgent data is being sent *)
  * (* stop persisting *)
else
  (* Otherwise, leave the timers alone *)
  cb.tt_rexmt in

```

(* Time this segment if it is sensible to do so, i.e. the following conditions hold : (a) a segment is not already being timed, and (b) data or a FIN are being sent, and (c) the segment being emitted is not a retransmit, and (d) the segment is not a window probe *)

```

let t_rttseg' = if IS_NONE cb.t_rttseg ∧ (data_to_send ≠ [] ∨ FIN) ∧
  snd_nxt'' > cb.snd_max ∧ ¬window_probe
then
  ↑(ts_val', snd_nxt')
else
  cb.t_rttseg in

```

```
(* Update the socket *)
sock' = sock ⟨⟨ pr := TCP_PROTO(tcp_sock
  ⟨⟨ st := st'; cb := tcp_sock.cb
    ⟨⟨ tt_rexmt := tt_rexmt';
      snd_cwnd := snd_cwnd';
      rcv_wnd := rcv_wnd';
      tf_rxwin0sent := (rcv_wnd' = 0);
      tf_shouldacknow := F;
      t_rttseg := t_rttseg';
      snd_max := snd_max';
      snd_nxt := snd_nxt'';
      tt_delack := *;
      last_ack_sent := cb.rcv_nxt;
      rcv_adv := cb.rcv_nxt + rcv_wnd'
    ⟩⟩⟩⟩ ^
```

```
(* Constrain the list of output segments to contain just the segment being emitted *)
outsegs' = [TCP seg]
```

Description

This function constructs the next segment to be output. It is usually called once `tcp_output_required` has returned true, but sometimes is called directly when we wish always to emit a segment. A large number of TCP state variables are modified also.

Note that while constructing the segment a variety of errors such as ENOBUFS are possible, but this is not modelled here. Also, window shrinking is not dealt with properly here.

```
– combination of tcp_output_required and tcp_output_really :
tcp_output_perhaps arch ts_val ifds0 sock(sock', outsegs) =
let (do_output, persist_fun) = tcp_output_required arch ifds0 sock in
let sock'' =
option_case sock (λf.sock ⟨⟨ pr := TCP_PROTO(tcp_sock_of sock cb := f)⟩⟩) persist_fun in
if do_output then
tcp_output_really arch F ts_val ifds0 sock''(sock', outsegs)
else
(sock' = sock'' ∧ outsegs = [])
```

14.3 Segment Queuing (TCP only)

Once a segment is generated for output, it must be enqueued for transmission. This enqueueing may fail. These functions model what happens in this case, and encapsulate the enqueueing-and-possibly-rolling-back process.

14.3.1 Summary

| | |
|---|--|
| <code>rollback_tcp_output</code> | Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails |
| <code>enqueue_or_fail</code> | wrap <code>rollback_tcp_output</code> together with enqueue |
| <code>enqueue_or_fail_sock</code> | version of <code>enqueue_or_fail</code> that works with sockets rather than cbs |
| <code>enqueue_and_ignore_fail</code> | version of <code>enqueue_or_fail</code> that ignores errors and doesn't touch the <code>tcpcb</code> |
| <code>enqueue_each_and_ignore_fail</code> | version of above that ignores errors and doesn't touch the <code>tcpcb</code> |

mlift_tcp_output_perhaps_or_fail

do mliftc for function returning at most one segment and not dealing with queueing flag

14.3.2 Rules

– **Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails :**
 rollback_tcp_output rcvdsyn seg arch rttab ifds is_connect cb₀ cb_in(cb', es', outsegs') =

(* NB: from cb₀, only snd_nxt, tt_delack, last_ack_sent, rcv_adv, tf_rxwin0sent, t_rttseg, snd_max, tt_rexmt are used. *)

(choose allocated :: (if INFINITE_RESOURCES then {T} else {T; F})).

let route = test_outroute(seg, rttab, ifds, arch) in

let f0 = λcb.cb ⟨ (* revert to original values; on ip_output failure *)

 snd_nxt := cb₀.snd_nxt;

 tt_delack := cb₀.tt_delack;

 last_ack_sent := cb₀.last_ack_sent;

 rcv_adv := cb₀.rcv_adv

⟩ in

let f1 = λcb.if ¬rcvdsyn then

 cb

else

 cb ⟨ (* set soft error flag; on ip_output routing failure *)

 t_softerror := **the** route(* assumes route = SOME (SOME e) *)

 ⟩ in

let f2 = λcb.cb ⟨ (* revert to original values; on early ENOBUFS *)

 tf_rxwin0sent := cb₀.tf_rxwin0sent;

 t_rttseg := cb₀.t_rttseg;

 snd_max := cb₀.snd_max;

 tt_rexmt := cb₀.tt_rexmt

⟩ in

let f3 = λcb.if **is_some** cb.tt_rexmt ∨ is_connect then (* quench; on ENOBUFS *)

 cb

else

 cb ⟨ (* maybe start rexmt and close down window *)

 tt_rexmt := start_tt_rexmt arch 0 **F** cb.t_rttinf;

 snd_cwnd := cb.t_maxseg(* no LAN allowance, by design *)

 ⟩ in

if ¬allocated then (* allocation failure *)

 cb' = f3(f2(f0 cb_in)) ∧ outsegs' = [] ∧ es' = ↑ ENOBUFS

else if route = * then (* ill-formed segment *)

 ASSERTION_FAILURE“rollback_tcp_output:1”(* should never happen *)

else if ∃e.route = ↑(↑ e) then (* routing failure *)

 cb' = f1(f0 cb_in) ∧ outsegs' = [] ∧ es' = **the** route

else if loopback_on_wire seg ifds then (* loopback not allowed on wire - RFC1122 *)

 (if windows_arch arch then

 cb' = cb_in ∧ outsegs' = [] ∧ es' = (* Windows silently drops segment! *)

 else if bsd_arch arch then

 cb' = f0 cb_in ∧ outsegs' = [] ∧ es' = ↑ EADDRNOTAVAIL

 else if linux_arch arch then

 cb' = f0 cb_in ∧ outsegs' = [] ∧ es' = ↑ EINVAL

 else

 ASSERTION_FAILURE“rollback_tcp_output:2”(* never happen *)

)

else

 (∃queued.

```

    outsegs' = [(seg, queued)] ∧
    if ¬queued then (* queueing failure *)
        cb' = f3(f0 cb_in) ∧ es' = ↑ ENOBUFFS
    else (* success *)
        cb' = cb_in ∧ es' = *)
)

```

– **wrap rollback_tcp_output together with enqueue :**

```

enqueue_or_fail rcvdsyn arch rttab ifds outsegs oq cb_0 cb_in(cb', oq') =
(case outsegs of
[] → cb' = cb_0 ∧ oq' = oq
|| [seg] → (∃outsegs' es'.
    rollback_tcp_output rcvdsyn seg arch rttab ifds F cb_0 cb_in(cb', es', outsegs') ∧
    enqueue_oq_list_qinfo(oq, outsegs', oq'))
|| _other84 → ASSERTION_FAILURE“enqueue_or_fail”(* only 0 or 1 segments at a time *)
)

```

– **version of enqueue_or_fail that works with sockets rather than cbs :**

```

enqueue_or_fail_sock rcvdsyn arch rttab ifds outsegs oq sock0 sock(sock', oq') =
(* NB: could calculate rcvdsyn, but clearer to pass it in *)
let tcp_sock = tcp_sock_of sock in
let tcp_sock0 = tcp_sock_of sock0 in
(∃cb'.
enqueue_or_fail rcvdsyn arch rttab ifds outsegs oq(tcp_sock_of sock0).cb(tcp_sock_of sock).cb(cb', oq') ∧
sock' = sock ⟨ pr := TCP_PROTO(tcp_sock_of sock ⟨
    cb := cb'
    ⟩)⟩)
)

```

– **version of enqueue_or_fail that ignores errors and doesn't touch the tcpcb :**

```

enqueue_and_ignore_fail arch rttab ifds outsegs oq oq' =
∃rcvdsyn cb_0 cb_in cb'.
enqueue_or_fail rcvdsyn arch rttab ifds outsegs oq cb_0 cb_in(cb', oq')

```

– **version of above that ignores errors and doesn't touch the tcpcb :**

```

(enqueue_each_and_ignore_fail arch rttab ifds[] oq oq' = (oq = oq')) ∧
(enqueue_each_and_ignore_fail arch rttab ifds(seg :: segs) oq oq''
= ∃oq'. enqueue_and_ignore_fail arch rttab ifds[seg] oq oq' ∧
    enqueue_each_and_ignore_fail arch rttab ifds segs oq' oq'')

```

– **do mliftc for function returning at most one segment and not dealing with queueing flag :**

```

mlift_tcp_output_perhaps_or_fail ts_val arch rttab ifds_0 =
mliftc(λs(s', outsegs').
    ∃s_1 segs.
    tcp_output_perhaps arch ts_val ifds_0 s(s_1, segs) ∧
    case segs of

```

```

[] → s' = s1 ∧ outsegs' = []
|| [seg] → (∃cb' es'.(* ignore error return *)
            rollback_tcp_output T seg arch rttab ifds0 F
            (tcp_sock_of s).cb(tcp_sock_of s1).cb(cb', es', outsegs') ∧
            s' = s1 ⟨ pr := TCP_PROTO(tcp_sock_of s1 ⟨ cb := cb' ⟩) ⟩)
|| _other58 → ASSERTION_FAILURE“mlift_tcp_output_perhaps_or_fail”(* never happen *)
)

```

14.4 Incoming Segment Functions (TCP only)

Updates performed to the idle, keepalive, and FIN_WAIT_2 timers for every incoming segment.

14.4.1 Summary

update_idle Do updates appropriate to receiving a new segment on a connection

14.4.2 Rules

– Do updates appropriate to receiving a new segment on a connection :

```

update_idle tcp_sock =
let t_idletime' = stopwatch_zero in (* update 'time most recent packet received' field *)
let tt_keep' = (if ¬(tcp_sock.st = SYN_RECEIVED ∧ tcp_sock.cb.tf_needfin) then
                (* reset keepalive timer to 2 hours. *)
                ↑((( )slow_timer TCPTV_KEEP_IDLE)
                else
                tcp_sock.cb.tt_keep) in
let tt_fin_wait_2' = (if tcp_sock.st = FIN_WAIT_2 then
                    ↑((( )slow_timer TCPTV_MAXIDLE)
                    else
                    tcp_sock.cb.tt_fin_wait_2) in
(t_idletime', tt_keep', tt_fin_wait_2')

```

14.5 Drop Segment Functions (TCP only)

When an erroneous or unexpected segment arrives, it is usually dropped (i.e, ignored). However, the peer is usually informed immediately by means of a RST or ACK segment.

14.5.1 Summary

dropwithreset emit a RST segment corresponding to the passed segment, unless that would be stupid.

mlift_dropafterack_or_fail send immediate ACK to segment, but otherwise process it no further

dropwithreset_ignore_fail do emit_segs_pred, for function returning at most one seg and not dealing with queueing flag

14.5.2 Rules

– **emit a RST segment corresponding to the passed segment, unless that would be stupid.** :

dropwithreset *seg ifds₀ ticks reason bndlm bndlm' outsegs* =

(* Needs list of the host's interfaces, to verify that the incoming segment wasn't broadcast. Returns a list of segments. *)

if (* never RST a RST *)

seg.RST ∨

(* is segment a (link-layer?) broadcast or multicast? *)

F ∨

(* is source or destination broadcast or multicast? *)

(∃*i*₁.*seg.is*₁ = ↑ *i*₁ ∧ is_broadormulticast ∅ *i*₁) ∨

(∃*i*₂.*seg.is*₂ = ↑ *i*₂ ∧ is_broadormulticast *ifds*₀ *i*₂)

(* BSD only checks incoming interface, but should have same effect as long as interfaces don't overlap *)

then

outsegs = [] ∧ *bndlm'* = *bndlm*

else

(**choose** *seg'* :: make_rst_segment_from_seg *seg*.)

let (*emit*, *bndlm''*) = bandlim_rst_ok(*seg'*, *ticks*, *reason*, *bndlm*) **in** (* finally: check if band-limited *)

bndlm' = *bndlm''* ∧

outsegs = **if** *emit* **then** [TCP *seg'*] **else** [])

– **send immediate ACK to segment, but otherwise process it no further** :

mlift_dropafterack_or_fail *seg arch rttab ifds ticks(sock, bndlm)((sock', bndlm', outsegs'), continue)* =

(* *ifds* is just in case we need to send a RST, to make sure we don't send it to a broadcast address. *)

let *tcp_sock* = tcp_sock_of *sock* **in**

(*continue* = **T** ∧

let *cb* = *tcp_sock.cb* **in**

if *tcp_sock.st* = SYN_RECEIVED ∧

seg.ACK ∧

(**let** *ack* = *tcp_seq_flip_sense seg.ack* **in**

(*ack* < *cb.snd_una* ∨ *cb.snd_max* < *ack*))

then

(* break loop in "LAND" DoS attack, and also prevent ACK storm between two listening ports that have been sent forged SYN segments, each with the source address of the other. (tcp_input.c:2141) *)

sock' = *sock* ∧

dropwithreset *seg ifds ticks BANDLIM_RST_OPENPORT bndlm bndlm'(map fst outsegs')*

(* ignore queue full error *)

else

(∃*sock*₁ *msg cb' es'*.(* ignore errors *))

let *tcp_sock1* = tcp_sock_of *sock*₁ **in**

tcp_output_really *arch F ticks ifds sock(sock₁, [msg])* ∧ (* did set *tf_acknow* and call *tcp_output_perhaps*, which seemed a bit silly *)

(* notice we here bake in the assumption that the timestamps use the same counter as the band limiter; perhaps this is unwise *)

rollback_tcp_output **T msg arch rttab ifds F tcp_sock.cb tcp_sock1.cb(cb', es', outsegs')** ∧

sock' = *sock*₁ [*pr* := TCP_PROTO(*tcp_sock1* [*cb* := *cb'*])] ∧

bndlm' = *bndlm*)

– **do emit_segs_pred, for function returning at most one seg and not dealing with queueing flag** :

dropwithreset_ignore_fail *seg_in arch ifds rttab ticks reason b b'(outsegs' : (msg#bool)list)* =

```

(* No rollback necessary here. *)
∃ segs.
dropwithreset seg_in ifds ticks reason b b' segs ∧
case segs of
  [] → outsegs' = []
  || [seg] → (choose allocated :: if INFINITE_RESOURCES then {T} else {T; F}.
    if ¬allocated then
      outsegs' = []
    else
      (case test_outroute(seg, rttab, ifds, arch) of
        * → ASSERTION_FAILURE“dropwithreset_ignore_fail:1”(* never happen *)
        || ↑(↑ e) → outsegs' = [](* ignore error *)
        || ↑ * → ∃ queued.outsegs' = [(seg, queued)]))
  || _other57 → ASSERTION_FAILURE“dropwithreset_ignore_fail:2”(* never happen *)

```

14.6 Close Functions (TCP only)

Closing a connection, updating the socket and TCP control block appropriately.

14.6.1 Summary

| | |
|---------------------------|---|
| <i>tcp_close</i> | close the socket and remove the TCPCB |
| <i>tcp_drop_and_close</i> | drop TCP connection, reporting the specified error. If synchronised, send RST to peer |

14.6.2 Rules

– **close the socket and remove the TCPCB :**

```

tcp_close arch sock = sock
⟦ cantrcvmore := T; (* MF doesn't believe this is correct for Linux or WinXP *)
  cantsndmore := T;
  is1 := if bsd_arch arch then * else sock.is1;
  ps1 := if bsd_arch arch then * else sock.ps1;
  pr := TCP_PROTO(tcp_sock_of sock
  ⟦ st := CLOSED;
    cb := initial_cb (* in reality, it's dropped entirely, but we don't do that *)
    ⟦ bsd_cantconnect := if bsd_arch arch then T else F];
    sndq := [] ⟩
  ⟩

```

Description This is similar to BSD's `tcp_close()`, except that we do not actually remove the protocol/control blocks. The quad of the socket is cleared, to enable another socket to bind to the port we were previously using — this isn't actually done by BSD, but the effect is the same. The `bsd_cantconnect` flag is set to indicate that the socket is in such a detached state.

– **drop TCP connection, reporting the specified error. If synchronised, send RST to peer :**

```

tcp_drop_and_close arch err sock (sock', outsegs) =
let tcp_sock = tcp_sock_of sock in (
if tcp_sock.st ∉ {CLOSED; LISTEN; SYN_SENT} then

```

```

    (choose seg :: (make_rst_segment_from_cb tcp_sock.cb
                    (the sock.is1, the sock.is2, the sock.ps1, the sock.ps2)).
    outsegs = [TCP seg])
  else
    outsegs = [] ^
let es' =
if err = ↑ ETIMEDOUT then
  (if tcp_sock.cb.t_softerror ≠ * then
    tcp_sock.cb.t_softerror
  else
    ↑ ETIMEDOUT)
else if err ≠ * then err
else sock.es
in
sock' = tcp_close arch(sock ⟨ es := es' ⟩))

```

Description BSD calls this tcp_drop

Part XIII

TCP1_hostLTS

Chapter 15

Host LTS: Socket Calls

15.1 accept() (TCP only)

accept : fd → fd * (ip * port)

accept(fd) returns the next connection available on the completed connections queue for the listening TCP socket referenced by file descriptor fd. The returned file descriptor fd refers to the newly-connected socket; the returned ip and port are its remote address. accept() blocks if the completed connections queue is empty and the socket does not have the O_NONBLOCK flag set.

Any pending errors on the new connection are ignored, except for ECONNABORTED which causes accept() to fail with ECONNABORTED.

Calling accept() on a UDP socket fails: UDP is not a connection-oriented protocol.

15.1.1 Errors

A call to accept() can fail with the errors below, in which case the corresponding exception is raised:

| | |
|--------------|--|
| EAGAIN | The socket has the O_NONBLOCK flag set and no connections are available on the completed connections queue. |
| ECONNABORTED | The connection at the head of the completed connections queue has been aborted; the socket has been shutdown for reading; or the socket has been closed. |
| EINVAL | This socket is not accepting connections, i.e., it is not in the LISTEN state, or is a UDP socket. |
| EMFILE | The maximum number of file descriptors allowed per process are already open for this process. |
| EOPNOTSUPP | The socket type of the specified socket does not support accepting connections. This error is raised if accept() is called on a UDP socket. |
| ENFILE | Out of resources. |
| ENOBUFS | Out of resources. |
| ENOMEM | Out of resources. |
| EINTR | The system was interrupted by a caught signal. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.1.2 Common cases

accept() is called and immediately returns a connection: *accept_1; return_1*

accept() is called and blocks; a connection is completed and the call returns: *accept_2; deliver_in_99; deliver_in_1; accept_1; return_1*

15.1.3 API

```
Posix:      int accept(int socket, struct sockaddr *restrict address,
                    socklen_t *restrict address_len);
FreeBSD:    int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
Linux:      int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
WinXP:      SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen);
```

In the Posix interface:

- **socket** is the listening socket's file descriptor, corresponding to the `fd` argument of the model;
- the returned **int** is either non-negative, i.e., a file descriptor referring to the newly-connected socket, or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `INVALID_SOCKET`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.
- **address** is a pointer to a `sockaddr` structure of length `address_len` corresponding to the `ip * port` returned by the model `accept()`. If **address** is not a null pointer then it stores the address of the peer for the accepted connection. For the model `accept()` it will actually be a `sockaddr_in` structure; the peer IP address will be stored in the `sin_addr.s_addr` field, and the peer port will be stored in the `sin_port` field. If **address** is a null pointer then the peer address is ignored, but the model `accept()` always returns the peer address. On input the `address_len` is the length of the `address` structure, and on output it is the length of the stored address.

15.1.4 Model details

If the `accept()` call blocks then state `ACCEPT2(sid)` is entered, where `sid` is the index of the socket that `accept()` was called upon.

The following errors are not included in the model:

- **EFAULT** signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to `accept()` that is excluded by the clean interface used in the model.
- **EPERM** is a Linux-specific error code described by the Linux man page as "Firewall rules forbid connection". This is outside the scope of what is modelled.
- **EPROTO** is a Linux-specific error code described by the man page as "Protocol error". Only TCP and UDP are modelled here; the only sockets that can exist in the model are bound to a known protocol.
- **WSAECONNRESET** is a WinXP-specific error code described in the MSDN page as "An incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call." This error has not been encountered in exhaustive testing.
- **WSAEINPROGRESS** is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

From the Linux man page: Linux `accept()` passes already-pending network errors on the new socket as an error code from `accept`. This behaviour differs from other BSD socket implementations. For reliable operation the application should detect the network errors defined for the protocol after `accept` and treat them like `EAGAIN` by retrying. In case of TCP/IP these are `ENETDOWN`, `EPROTO`, `ENOPROTOOPT`, `EHOSTDOWN`, `ENONET`, `EHOSTUNREACH`, `EOPNOTSUPP`, and `ENETUNREACH`.

This is currently not modelled, but will be looked at when the Linux semantics are investigated.

15.1.5 Summary

| | | |
|-----------------|-----------------------|---|
| <i>accept_1</i> | tcp: rc | Return new connection; either immediately or from a blocked state. |
| <i>accept_2</i> | tcp: block | Block waiting for connection |
| <i>accept_3</i> | tcp: fast fail | Fail with EAGAIN: no pending connections and non-blocking semantics set |
| <i>accept_4</i> | tcp: rc | Fail with ECONNABORTED: the listening socket has <i>cantsndmore</i> set or has become CLOSED. Returns either immediately or from a blocked state. |
| <i>accept_5</i> | tcp: rc | Fail with EINVAL: socket not in LISTEN state |
| <i>accept_6</i> | tcp: rc | Fail with EMFILE: out of file descriptors |
| <i>accept_7</i> | udp: fast fail | Fail with EOPNOTSUPP or EINVAL: accept() called on a UDP socket |

15.1.6 Rules

accept_1 **tcp: rc** Return new connection; either immediately or from a blocked state.

$h \llbracket ts := ts \oplus (tid \mapsto (t)_d);$
 $fds := fds;$
 $files := files;$
 $socks :=$
 $socks \oplus$
 $[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,$
 $\text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis,$
 $[], *, [], *, \text{NO_OOBDATA}));$
 $(sid', \text{SOCK}(*, sf', \uparrow i'_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es', cantsndmore', cantrcvmore',$
 $\text{TCP_Sock}(\text{ESTABLISHED}, cb', *, sndq, sndurp, rcvq, rcvurp, iobc)))] \rrbracket$

$\xrightarrow{lbl} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(fd', (i_2, p_2))))_{\text{sched_timer}});$
 $fds := fds';$
 $files := files \oplus [(fid', \text{FILE}(\text{FT_SOCKET}(sid'), \text{ff_default}))];$
 $socks :=$
 $socks \oplus$
 $[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,$
 $\text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis',$
 $[], *, [], *, \text{NO_OOBDATA}));$
 $(sid', \text{SOCK}(\uparrow fid', sf', \uparrow i'_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es',$
 $cantsndmore', cantrcvmore', \text{TCP_Sock}(\text{ESTABLISHED}, cb', *, sndq,$
 $sndurp, rcvq, rcvurp, iobc)))] \rrbracket$

$$\left(\left(\begin{array}{l} t = \text{RUN} \wedge \\ lbl = tid \cdot (\text{accept } fd) \wedge \\ rc = \text{FAST SUCCEED} \wedge \\ fid = fds[fd] \wedge \\ fd \in \text{dom}(fds) \wedge \\ files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), \text{ff}) \end{array} \right) \vee \left(\begin{array}{l} t = \text{ACCEPT2}(sid) \wedge \\ lbl = \tau \wedge \\ rc = \text{SLOW urgent SUCCEED} \end{array} \right) \right) \wedge$$

$lis.q = q @ [sid'] \wedge$
 $lis'.q = q \wedge$
 $lis'.q_0 = lis.q_0 \wedge lis'.qlimit = lis.qlimit \wedge$
 $(sid \neq sid') \wedge$
 $es' \neq \uparrow \text{ECONNABORTED} \wedge$
 $fid' \notin ((\text{dom}(files)) \cup \{fid\}) \wedge$
 $\text{nextfd } h.\text{arch } fds \text{ } fd' \wedge$
 $fds' = fds \oplus (fd', fid') \wedge$

$$(\forall i_1. \uparrow i_1 = is_1 \implies i_1 = i'_1)$$

Description

This rule covers two cases: (1) the completed connection queue is non-empty when `accept(fd)` is called from a thread `tid` in the RUN state, where `fd` refers to a TCP socket `sid`, and (2) a previous call to `accept(fd)` on socket `sid` blocked, leaving its calling thread `tid` in state `ACCEPT2(sid)`, and a new connection has become available.

In either case the listening TCP socket `sid` has a connection `sid'` at the head of its completed connections queue `sid' :: q`. A socket entry for `sid'` already exists in the host's finite map of sockets, `socks ⊕ . . .`. The socket is ESTABLISHED, is not shutdown for reading, and is only missing a file description association that would make it accessible via the sockets interface.

A new file description record is created for connection `sid'`, indexed by a new `fd'`, and this is added to the host's finite map of file descriptions `files`. It is assigned a default set of file flags, `ff_default`. The socket entry `sid'` is completed with its file association $\uparrow fd'$ and `sid'` is removed from the head of the completed connections queue.

When the listening socket `sid` is bound to a local IP address `i1`, the accepted socket `sid'` is also bound to it.

Finally, the new file descriptor `fd'` is created in an architecture-specific way using the auxiliary `nextfd (p??)`, and an entry mapping `fd'` to `fd'` is added to the host's finite map of file descriptors. If the calling thread was previously blocked in state `ACCEPT2(sid)` it proceeds via a τ transition, otherwise by a `tid · (accept fd)` transition. The thread is left in state `RET(OK(fd', (i2, p2)))` to return the file descriptor and remote address of the accepted connection in response to the original `accept()` call.

If the new socket `sid'` has error `ECONNABORTED` pending in its error field `es'`, this is handled by rule `accept_5`. All other pending errors on `sid'` are ignored, but left as the socket's pending error.

accept_2 **tcp: block** **Block waiting for connection**

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid \cdot (\text{accept } fd)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{ACCEPT2}(sid))_{\text{never_timer}}) \rangle \rangle$$

$$\begin{aligned} & fd \in \mathbf{dom}(h.fds) \wedge \\ & fd = h.fds[fd] \wedge \\ & h.files[fd] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge \\ & ff.b(\mathbf{O_NONBLOCK}) = \mathbf{F} \wedge \\ & (\exists sf \ i_1 \ p_1 \ cb \ lis \ es. \\ & \quad h.socks[sid] = \mathbf{SOCK}(\uparrow fd, sf, i_1, \uparrow p_1, *, *, es, \mathbf{F}, \mathbf{cantrcvmore}, \\ & \quad \quad \quad \mathbf{TCP_Sock}(\mathbf{LISTEN}, cb, \uparrow lis, [], *, [], *, \mathbf{NO_OOBDATA})) \wedge \\ & \quad lis.q = [] \end{aligned}$$

Description

A blocking `accept()` call is performed on socket `sid` when no completed incoming connections are available. The calling thread blocks until a new connection attempt completes successfully, the call is interrupted, or the process runs out of file descriptors.

From thread `tid`, which is initially in the RUN state, `accept(fd)` is called where `fd` refers to listening TCP socket `sid` which is bound to local port `p1`, is not shutdown for reading and is in blocking mode: `ff.b(O_NONBLOCK) = F`. The socket's queue of completed connections is empty, `q := []`, hence the `accept()` call blocks waiting for a successful new connection attempt, leaving the calling thread state `ACCEPT2(sid)`.

Socket `sid` might not be bound to a local IP address, i.e. `i1` could be `*`. In this case the socket is listening for connection attempts on port `p1` for all local IP addresses.

accept_3 **tcp: fast fail** Fail with **EAGAIN**: no pending connections and non-blocking semantics set

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot (\text{accept } fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned} & fd \in \mathbf{dom}(h.fds) \wedge \\ & h.fds[fd] = fid \wedge \\ & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ & ff.b(\text{O_NONBLOCK}) = \mathbf{T} \wedge \\ & (\exists sf \ is_1 \ p_1 \ cb \ lis \ es. \\ & h.socks[sid] = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \\ & \quad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, [], *, [], *, \text{NO_OOBDATA})) \wedge \\ & lis.q = [] \end{aligned}$$

Description

A non-blocking `accept()` call is performed on socket `sid` when no completed incoming connections are available. Error `EAGAIN` is returned to the calling thread.

From thread `tid`, which is initially in the `RUN` state, `accept(fd)` is called where `fd` refers to a listening TCP socket `sid` which is bound to local port `p1`, not shutdown for writing, and in non-blocking mode: `ff.b(O_NONBLOCK) = T`. The socket's queue of completed connections is empty, `q := []`, hence the `accept()` call returns error `EAGAIN`, leaving the calling thread state `RET(FAIL EAGAIN)` after a `tid-accept(fd)` transition.

Socket `sid` might not be bound to a local IP address, i.e. `is1` could be `*`. In this case the socket is listening for connection attempts on port `p1` for all local IP addresses.

accept_4 **tcp: rc** Fail with **ECONNABORTED**: the listening socket has `cantsndmore` set or has become **CLOSED**. Returns either immediately or from a blocked state.

$$\begin{aligned} & h \langle ts := ts \oplus (tid \mapsto (t)_d); \\ & socks := \\ & socks \oplus \\ & [(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \\ & \quad \text{TCP_Sock}(st, cb, \uparrow lis, [], *, [], *, \text{NO_OOBDATA})))] \rangle \\ \xrightarrow{lbl} & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ECONNABORTED}))_{\text{sched_timer}}); \\ & socks := \\ & socks \oplus \\ & [(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \\ & \quad \text{TCP_Sock}(st, cb, \uparrow lis, [], *, [], *, \text{NO_OOBDATA})))] \rangle \end{aligned}$$

$$\left(\begin{array}{l} t = \text{RUN} \wedge \\ st = \text{LISTEN} \wedge \\ \text{cantsndmore} = \mathbf{T} \wedge \\ lbl = tid \cdot \text{accept}(fd) \wedge \\ rc = \text{FAST FAIL} \wedge \\ fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \end{array} \right) \vee \left(\begin{array}{l} t = \text{ACCEPT2}(sid) \wedge \\ ((\text{cantrcvmore} = \mathbf{T} \wedge st = \text{LISTEN}) \vee \\ (st = \text{CLOSED})) \wedge \\ lbl = \tau \wedge \\ rc = \text{SLOW urgent FAIL} \end{array} \right)$$

Description

This rule covers two cases: (1) an `accept(fd)` call is made on a listening TCP socket `sid`, referenced by `fd`, with `cantsndmore` set, and (2) a previous call to `accept()` on socket `sid` blocked, leaving a thread `tid` in state `ACCEPT2(sid)`, but the socket has since either entered the `CLOSED` state, or had `cantrcvmore` set. In both cases, `ECONNABORTED` is returned.

This situation will arise only when a thread calls `close()` on the listening socket while another thread is blocking on an `accept()` call, or if `listen()` was originally called on a socket which already had `cantrcvmore` set. The latter can occur in BSD, which allows `listen()` to be called in any (non CLOSED or LISTEN) state, though should never happen under typical use.

If the calling thread was previously blocked in state `ACCEPT2(sid)`, it proceeds via an τ transition, otherwise by a $tid \cdot \text{accept}(fd)$ transition. The thread is left in state `RET(FAIL ECONNABORTED)` to return the error `ECONNABORTED` in response to the initial `accept()` call.

Note that this rule is not correct when dealing with the FreeBSD behaviour which allows any socket to be placed in the LISTEN state.

accept_5 **tcp: rc** **Fail with EINVAL: socket not in LISTEN state**

$$h \langle ts := ts \oplus (tid \mapsto (t)_d) \rangle \xrightarrow{lbl} h \langle ts := ts \oplus (tid \mapsto (\text{RET(FAIL EINVAL)})_{\text{sched_timer}}) \rangle$$

$$\left(\left(\begin{array}{l} t = \text{RUN} \wedge \\ lbl = tid \cdot \text{accept}(fd) \wedge \\ rc = \text{FAST FAIL} \wedge \\ fd \in \text{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fd] = \text{FILE(FT_SOCKET}(sid), ff) \end{array} \right) \vee \left(\begin{array}{l} t = \text{ACCEPT2}(sid) \wedge \\ lbl = \tau \wedge \\ rc = \text{SLOW urgent FAIL} \end{array} \right) \right) \wedge$$

$$\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge \\ tcp_sock.st \neq \text{LISTEN}$$

Description

It is not valid to call `accept()` on a socket that is not in the LISTEN state.

This rule covers two cases: (1) on the non-listening TCP socket `sid`, `accept()` is called from a thread `tid`, which is in the RUN state, and (2) a previous call to `accept()` on TCP socket `sid` blocked because no completed connections were available, leaving thread `tid` in state `ACCEPT2(sid)` and after the `accept()` call blocked the socket changed to a state other than LISTEN.

In the first case the `accept(fd)` call on socket `sid`, referenced by file descriptor `fd`, proceeds by a $tid \cdot \text{accept}(fd)$ transition and in the latter by a τ transition. In either case, the thread is left in state `RET(FAIL ECONNABORTED)` to return error `ECONNABORTED` to the caller.

The second case is subtle: a previous call to `accept()` may have blocked waiting for a new completed connection to arrive and an operation, such as a `close()` call, in another thread caused the socket to change from the LISTEN state.

accept_6 **tcp: rc** **Fail with EMFILE: out of file descriptors**

$$h \langle ts := ts \oplus (tid \mapsto (t)_d) \rangle \xrightarrow{lbl} h \langle ts := ts \oplus (tid \mapsto (\text{RET(FAIL EMFILE)})_{\text{sched_timer}}) \rangle$$

$$\left(\left(\begin{array}{l} t = \text{RUN} \wedge \\ lbl = tid \cdot \text{accept}(fd) \wedge \\ rc = \text{FAST FAIL} \wedge \\ fd \in \text{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fd] = \text{FILE(FT_SOCKET}(sid), ff) \wedge \\ sock = (h.socks[sid]) \wedge \\ \text{proto_of } sock.pr = \text{PROTO_TCP} \end{array} \right) \vee \left(\begin{array}{l} t = \text{ACCEPT2}(sid) \wedge \\ lbl = \tau \wedge \\ rc = \text{SLOW nonurgent FAIL} \end{array} \right) \right) \wedge$$

$$\text{card}(\text{dom}(h.fds)) \geq \text{OPEN_MAX}$$

Description

This rule covers two cases: (1) from thread tid , which is in the RUN state, an `accept(fd)` call is made where fd refers to a TCP socket sid , and (2) a previous call to `accept()` blocked leaving thread tid in the `ACCEPT2(sid)` state. In either case the `accept()` call fails with `EMFILE` as the process (see Model Details) already has open its maximum number of open file descriptors `OPEN_MAX`.

In the first case the error is returned immediately (FAST FAIL) by performing an $tid\text{-}accept(fd)$ transition, leaving the thread state `RET(FAIL EMFILE)`. In the second, the thread is unblocked, also leaving the thread state `RET(FAIL EMFILE)`, by performing a τ transition.

Model details

In real systems, error `EMFILE` indicates that the calling process already has `OPEN_MAX` file descriptors open and is not permitted to open any more. This specification only models one single-process host with multiple threads, thus `EMFILE` is generated when the host exceeds the `OPEN_MAX` limit in this model.

accept_7 **udp: fast fail** Fail with `EOPNOTSUPP` or `EINVAL`: `accept()` called on a UDP socket

$$h \langle ts := ts \oplus (tid \mapsto (RUN)_d) \rangle \xrightarrow{tid\text{-}accept(fd)} h \langle ts := ts \oplus (tid \mapsto (RET(FAIL\ err))_{\text{sched_timer}}) \rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge$
 $\mathbf{proto_of}(h.socks[sid]).pr = \mathbf{PROTO_UDP} \wedge$
(if `bsd_arch` $h.arch$ **then** $err = \mathbf{EINVAL}$
else $err = \mathbf{EOPNOTSUPP}$)

Description

Calling `accept()` on a socket for a connectionless protocol (such as UDP) has no defined behaviour and is thus an invalid (`EINVAL`) or unsupported (`EOPNOTSUPP`) operation.

From thread tid , which is in the RUN state, an `accept(fd)` call is made where fd refers to a UDP socket identified by sid . The call proceeds by a $tid\text{-}accept(fd)$ transition leaving the thread state `RET(FAIL err)` to return error err . On FreeBSD err is `EINVAL`; on all other systems the error is `EOPNOTSUPP`.

Variations

| | |
|---------|---|
| FreeBSD | FreeBSD returns error <code>EINVAL</code> if <code>accept()</code> is called on a UDP socket. |
|---------|---|

15.2 bind() (TCP and UDP)

`bind : (fd * ip option * port option) → unit`

`bind(fd, is, ps)` assigns a local address to the socket referenced by file descriptor fd . The local address, (is, ps) , may consist of an IP address, a port or both an IP address and port.

If `bind()` is called without specifying a port, `bind(-, -, *)`, the socket's local port assignment is autobound, i.e. an unused port for the socket's protocol in the host's ephemeral port range is selected and assigned to the socket. Otherwise the port p specified in the bind call, `bind(-, -, ↑ p)` forms part of the socket's local address.

On some architectures a range of port values are designated to be privileged, e.g. 0-1023 inclusive. If a call to `bind()` requests a port in this range and the caller does not have sufficient privileges the call will fail.

A `bind()` call may or may not specify the IP address. If an IP address is not specified, `bind(-, *, -)`, the socket's local IP address is set to `*` and it will receive segments or datagrams addressed to any of the host's local IP addresses and port p . Otherwise, the caller specifies a local IP address, `bind(-, ↑ i, -)`, the socket's local IP address is set to i , and it only receives segments or datagrams addressed to IP address i and port p .

A call to `bind()` may be unsuccessful if the requested IP address or port is unavailable to bind to, although in certain situations this can be overridden by setting the socket option `SO_REUSEADDR` appropriately: see `bound_port_allowed` (p85).

A socket can only be bound once: it is not possible to rebind it to a different port later. A `bind()` call is not necessary for every socket: sockets may be autobound to an ephemeral port when a call requiring a port binding is made, e.g. `connect()`.

15.2.1 Errors

A call to `bind()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|---------------|--|
| EACCES | The specified port is in the privileged port range of the host architecture and the current thread does not have the required privileges to bind to it. |
| EADDRINUSE | The specified address is in use by or conflicts with the address of another socket using the same protocol. The error may occur in the following situations only: <ul style="list-style-type: none"> • <code>bind(-, -, ↑ p)</code> will fail with EADDRINUSE if another socket is bound to port <i>p</i>. This error may be preventable by setting the SO_REUSEADDR socket option. • <code>bind(-, ↑ i, ↑ p)</code> will fail with EADDRINUSE if another socket is bound to port <i>p</i> and IP address <i>i</i>, or is bound to port <i>p</i> and wildcard IP. This error will not occur if the SO_REUSEADDR option is correctly used to allow multiple sockets to be bound to the same local port. <p>This error is never returned from a call <code>bind(-, -, *)</code> that requests an autobound port.</p> |
| EADDRNOTAVAIL | The specified IP address cannot be bound as it is not local to the host. |
| EINVAL | The socket is already bound to an address and the socket's protocol does not support rebinding to a new address. Multiple calls to <code>bind()</code> are not permitted. |
| EISCONN | The socket is connected and rebinding to a new local address is not permitted (TCP ONLY). |
| ENOBUFS | A port was not specified in the <code>bind()</code> call and autobinding failed because no ephemeral ports for the socket's protocol are currently available. In addition, on WinXP the error can signal that the host has insufficient available buffers to complete the operation. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.2.2 Common cases

A server application creates a TCP socket and binds it to its local address. It is then put in the LISTEN state to accept incoming connections to this address: `socket_1; return_1; bind_1; return_1; listen_1`

A UDP socket is created and bound to its local address. `recv()` is called and the socket blocks, waiting to receive datagrams sent to the local address: `socket_1; return_1; bind_1; return_1; recv_12`

15.2.3 API

```
Posix:    int bind(int socket, const struct sockaddr *address,
               socklen_t address_len);
FreeBSD:  int bind(int s, struct sockaddr *addr, socklen_t addrlen);
Linux:    int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
WinXP:    SOCKET bind(SOCKET s, const struct sockaddr* name, int namelen);
```

In the Posix interface:

- **socket** is the socket's file descriptor, corresponding to the `fd` argument of the model.
- **address** is a pointer to a `sockaddr` structure of size `socklen_t` containing the local IP address and port to be assigned to the socket, corresponding to the `is` and `ps` arguments of the model. For the `AF_INET` sockets used in the model, a `sockaddr_in` structure stores the address. The `sin_addr.s_addr` field holds the IP address; if it is set to 0 then the IP address is wildcarded: `is = *`. The `sin_port` field stores the port to bind to; if it is set to 0 then the port is wildcarded: `ps = *`. On WinXP a wildcard IP is specified by the constant `INADDR_ANY`, not 0
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo some argument renaming, except where noted above.

On Windows Socket 2 the `name` parameter is not necessarily interpreted as a pointer to a `sockaddr` structure but is cast this way for compatibility with Windows Socket 1.1 and the BSD sockets interface. The service provider implementing the functionality can choose to interpret the pointer as a pointer to any block of memory provided that the first two bytes of the block start with the address family used to create the socket. The default WinXP internet family provider expects a `sockaddr` structure here. This change is purely an interface design choice that ultimately achieves the same functionality of providing a name for the socket and is not modelled.

15.2.4 Model details

The specification only models the `AF_INET` address families thus the address family field of the `struct sockaddr` argument to `bind()` and those errors specific to other address families, e.g. UNIX domain sockets, are not modelled here.

In the Posix specification, `ENOBUFS` may have the additional meaning of "Insufficient resources were available to complete the call". This is more general than the use of `ENOBUFS` in the model.

The following errors are not modelled:

- `EAGAIN` is BSD-specific and described in the man page as: "Kernel resources to complete the request are temporarily unavailable". This is not modelled here.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `EFAULT` signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to `bind()` that is excluded by the clean interface used in the model. On WinXP, the equivalent error `WSAEFAULT` in addition signifies that the name address format used in `name` may be incorrect or the address family in `name` does not match that of the socket.
- `ENOTDIR`, `ENAMETOOLONG`, `ENOENT`, `ELOOP`, `EIO` (BSD-only), `EROFS`, `EISDIR` (BSD-only), `ENOMEM`, `EAFNOTSUPPORT` (Posix-only) and `EOPNOTSUPP` (Posix-only) are errors specific to other address families and are not modelled here. None apply to WinXP as other address families are not available by default.

15.2.5 Summary

| | | |
|---------------------|--------------------------|--|
| <code>bind_1</code> | all: fast succeed | Successfully assign a local address to a socket (possibly by autobinding the port) |
| <code>bind_2</code> | all: fast fail | Fail with <code>EADDRINUSE</code> : the specified address is already in use |
| <code>bind_3</code> | all: fast fail | Fail with <code>EADDRNOTAVAIL</code> : the specified IP address is not available on the host |
| <code>bind_5</code> | all: fast fail | Fail with <code>EINVAL</code> : the socket is already bound to an address and does not support rebinding; or socket has been shutdown for writing on FreeBSD |

| | | |
|---------------|--------------------------|---|
| <i>bind_7</i> | all: fast fail | Fail with EACCES: the specified port is privileged and the current process does not have permission to bind to it |
| <i>bind_9</i> | all: fast badfail | Fail with ENOBUFS: no ephemeral ports free for autobinding or, on WinXP only, insufficient buffers available. |

15.2.6 Rules

bind_1 **all: fast succeed** Successfully assign a local address to a socket (possibly by autobinding the port)

$$h_0 \xrightarrow{tid \cdot \text{bind}(fd, is_1, ps_1)} h$$

$$h_0 = h' \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$\text{socks} := \text{socks} \oplus$$

$$[(sid, \text{SOCK}(\uparrow fd, sf, *, *, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, pr))] \rangle \wedge$$

$$h = h' \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}});$$

$$\text{socks} := \text{socks} \oplus$$

$$[(sid, \text{SOCK}(\uparrow fd, sf, is_1, \uparrow p_1, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, pr))] \rangle;$$

$$\text{bound} := \text{bound} \rangle \wedge$$

$$fd \in \text{dom}(h_0.fds) \wedge$$

$$fd = h_0.fds[fd] \wedge$$

$$h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$sid \notin (\text{dom}(\text{socks})) \wedge$$

$$(\forall i_1. is_1 = \uparrow i_1 \implies i_1 \in \text{local_lips}(h_0.ifds)) \wedge$$

$$p_1 \in \text{autobind}(ps_1, (\text{proto_of } pr), \text{socks}) \wedge$$

$$\text{bound} = sid :: h_0.\text{bound} \wedge$$

$$(h_0.\text{privs} \vee p_1 \notin \text{privileged_ports}) \wedge$$

$$\text{bound_port_allowed } pr(h_0.\text{socks} \setminus \{sid\}) sf h_0.\text{arch } is_1 p_1 \wedge$$

(**case** *pr* **of**

$$\text{TCP_PROTO}(tcp_sock) \rightarrow tcp_sock = \text{TCP_Sock0}(\text{CLOSED}, cb, *, [], *, [], *, \text{NO_OOBDATA}) \wedge$$

$$(\text{bsd_arch } h_0.\text{arch} \implies \text{cantsndmore} = \mathbf{F} \wedge cb.\text{bsd_cantconnect} = \mathbf{F}) \parallel$$

$$\text{UDP_PROTO}(udp_sock) \rightarrow udp_sock = \text{UDP_Sock0}([])$$

Description

The call `bind(fd, is1, ps1)` is performed on the TCP or UDP socket *sid* referenced by file descriptor *fd* from a thread *tid* in the RUN state. The socket *sid* is currently uninitialised, i.e. it has no local or remote address defined (*, *, *, *), and it contains an uninitialised TCP or UDP protocol block, *tcp_sock* and *udp_sock* as appropriate for the socket's protocol.

If an IP address is specified in the `bind()` call, i.e. *is*₁ = $\uparrow i_1$, the call can only succeed if the IP address *i*₁ is one of those belonging to an interface of host *h*, *i*₁ \in `localLips(h0.ifds)`.

The port *p*₁ that the socket will be bound to is determined by the auxiliary function `autobind` that takes as argument the port option *ps*₁ from the `bind()` call. If *ps*₁ = $\uparrow p$ `autobind` simply returns the singleton set {*p*}, constraining the local port binding *p*₁ by *p*₁ = *p*. Otherwise, `autobind` returns a set of available ephemeral ports and *p*₁ is constrained to be a port within the set.

If a port is specified in the `bind()` call, i.e. *ps*₁ = $\uparrow p_1$, either the port is not a privileged port *p*₁ \notin `privileged_ports` or the host (actually, process) must have sufficient privileges *h*₀.*priv* = **T**.

Not all requested bindings are permissible because other sockets in the system may be bound to the chosen address or to a conflicting address. To check the binding *is*₁, $\uparrow p_1$ is permitted the auxiliary function `bound_port_allowed` is used. `bound_port_allowed` is architecture dependent and checks not only the other sockets bound locally to port *p*₁ on the host, but also the status of the socket flag `SO_REUSEADDR` for socket *sid* and the conflicting sockets. The use of the socket flag `SO_REUSEADDR` can permit sockets to share bindings under some circumstances, resolving the binding conflict. See `bound_port_allowed` (p85) for further information.

The call proceeds by performing a $tid\text{-}bind(fd, is_1, ps_1)$ transition returning $OK()$ to the calling thread. Socket sid is bound to local address $(is_1, \uparrow p_1)$ and the host has an updated list of bound sockets $bound$ with socket sid at its head.

Model details

The list of bound sockets $bound$ is used by the model to determine the order in which sockets are bound. This is required to model ICMP message and UDP datagram delivery on Linux.

Variations

| | |
|---------|--|
| FreeBSD | If sid is a TCP socket then it cannot be shutdown for writing: $cantsndmore = \mathbf{F}$, and its $bsd_cantconnect$ flag cannot be set. |
|---------|--|

bind_2 all: fast fail Fail with EADDRINUSE: the specified address is already in use

$$\frac{h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket}{tid\text{-}bind(fd, is_1, \uparrow p_1)} \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EADDRINUSE}))_{\text{sched_timer}}) \rrbracket$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &sock = (h.socks[sid]) \wedge \\ &\neg(\text{bound_port_allowed } sock.pr(h.socks \setminus sid) sock.sf h.arch is_1 p_1) \wedge \\ &(\mathbf{option_case } \mathbf{T} (\lambda i_1. i_1 \in \text{local_lips}(h.ifds)) is_1 \vee \text{windows_arch } h.arch) \end{aligned}$$

Description

From thread tid , which is in the RUN state, a $bind(fd, is_1, \uparrow p_1)$ call is performed on the socket $sock$, which is identified by sid and referenced by fd .

If an IP address is specified in the call, $is_1 = \uparrow i_1$, then i_1 must be an IP address for one of the host's interfaces. The requested local address binding, $(is_1, \uparrow p_1)$, is not available as it is already in use: see $\text{bound_port_allowed}$ (p85) for details.

The call proceeds by a $tid\text{-}bind(fd, is_1, \uparrow p_1)$ transition leaving the thread in state $\text{RET}(\text{FAIL EADDRINUSE})$ to return error EADDRINUSE to the caller.

bind_3 all: fast fail Fail with EADDRNOTAVAIL: the specified IP address is not available on the host

$$\frac{h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket}{tid\text{-}bind(fd, \uparrow i_1, ps_1)} \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EADDRNOTAVAIL}))_{\text{sched_timer}}) \rrbracket$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &i_1 \notin \text{local_lips}(h.ifds) \end{aligned}$$

Description

From thread tid , which is in the RUN state, a $bind(fd, \uparrow i_1, ps_1)$ call is made where fd refers to a socket sid .

The IP address, i_1 , to be assigned as part of the socket's local address does not belong to any of the interfaces on the host, $i_1 \notin \text{local_lips}(h.ifds)$, and therefore can not be assigned to the socket.

The call proceeds by a $tid\text{-}bind(fd, \uparrow i_1, ps_1)$ transition leaving the thread in state $\text{RET}(\text{FAIL EADDRNOTAVAIL})$ to return error EADDRNOTAVAIL to the caller.

bind_5 all: fast fail Fail with EINVAL: the socket is already bound to an address and does not support rebinding; or socket has been shutdown for writing on FreeBSD

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{bind}(fd, is_1, ps_1)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned} &fd \in \text{dom}(h.fds) \wedge \\ &fd = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &h.socks[sid] = sock \wedge \\ &(sock.ps_1 \neq * \vee \\ &(\text{bsd_arch } h.arch \wedge sock.pr = \text{TCP_PROTO}(tcp_sock) \wedge \\ & \quad (sock.cantsndmore \vee \\ & \quad tcp_sock.cb.bsd_cantconnect))) \end{aligned}$$

Description From thread tid , which is in the RUN state, a $\text{bind}(fd, is_1, ps_1)$ call is made where fd refers to a socket $sock$. The socket already has a local port binding: $sock.ps_1 \neq *$, and rebinding is not supported.

A $tid \cdot \text{bind}(fd, is_1, ps_1)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EINVAL})$.

Variations

| | |
|---------|---|
| FreeBSD | This rule also applies if fd refers to a TCP socket which is either shut down for writing or has its $bsd_cantconnect$ flag set. |
|---------|---|

bind_7 all: fast fail Fail with EACCES: the specified port is priveleged and the current process does not have permission to bind to it

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \quad tid \cdot \text{bind}(fd, is_1, \uparrow p_1)}{h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EACCES}))_{\text{sched_timer}}) \rangle}$$

$$\begin{aligned} &fd \in \text{dom}(h.fds) \wedge \\ &fd = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &(\neg h.privs \wedge p_1 \in \text{privileged_ports}) \end{aligned}$$

Description

From thread tid , which is in the RUN state, a $\text{bind}(fd, is_1, \uparrow p_1)$ call is made where fd refers to a socket sid . The port specified in the bind call, p_1 , lies in the host's range of privileged ports, $p_1 \in \text{privileged_ports}$, and the current host (actually, process) does not have sufficient permissions to bind to it: $\neg h.privs$.

The call proceeds by a $tid \cdot \text{bind}(fd, is_1, \uparrow p_1)$ transition leaving the thread in state $\text{RET}(\text{FAIL EACCES})$ to return the access violation error EACCES to the caller.

bind_9 all: fast badfail Fail with ENOBUFS: no ephemeral ports free for autobinding or, on WinXP only, insufficient buffers available.

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \quad tid \cdot \text{bind}(fd, is_1, ps_1)}{h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOBUFS}))_{\text{sched_timer}}) \rangle}$$

$$\begin{aligned} &fd \in \text{dom}(h.fds) \wedge \\ &fd = h.fds[fd] \wedge \end{aligned}$$

$$\begin{aligned}
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&ps_1 = * \wedge \\
&((\text{autobind}(ps_1, (\text{proto_of}(h.socks[sid]).pr), h.socks) = \emptyset) \vee \\
&\text{windows_arch } h.arch)
\end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a `bind(fd, is1, ps1)` call is made where *fd* refers to a socket *sid*.

A port is not specified in the `bind` call, i.e. $ps_1 = *$, and calling `autobind` returns the \emptyset set rather than a set of free ephemeral ports that the socket could choose from. This occurs only when there are no remaining ephemeral ports available for autobinding.

The call proceeds by a *tid*.`bind`(*fd*, *is₁*, *ps₁*) transition leaving the thread state `RET(FAIL ENOBUFS)` to return the out of resources error `ENOBUFS` to the caller.

Model details

Posix reports `ENOBUFS` to signify that "Insufficient resources were available to complete the call". This is not modelled here.

Variations

| | |
|-------|--|
| WinXP | On WinXP this error can occur non-deterministically when insufficient buffers are available. |
|-------|--|

15.3 close() (TCP and UDP)

`close` : $fd \rightarrow \text{unit}$

A call `close(fd)` closes file descriptor *fd* so that it no longer refers to a file description and associated socket. The closed file descriptor is made available for reuse by the process. If the file descriptor is the last file descriptor referencing a file description the file description itself is deleted and the underlying socket is closed. If the socket is a UDP socket it is removed.

It is important to note the distinction drawn above: only closing the last file descriptor of a socket has an effect on the state of the file description and socket.

The following behaviour may occur when closing the last file descriptor of a TCP socket:

- A TCP socket may have the `SO_LINGER` option set which specifies a maximum duration in seconds that a `close(fd)` call is permitted to block.
 - In the normal case the `SO_LINGER` option is not set, the `close` call returns immediately and asynchronously sends any remaining data and gracefully closes the connection.
 - If `SO_LINGER` is set to a non-zero duration, the `close(fd)` call will block while the TCP implementation attempts to successfully send any remaining data in the socket's send buffer and gracefully close the connection. If the sending of remaining data and the graceful close are successful within the set duration, `close(fd)` returns successfully, otherwise the linger timer expires, `close(fd)` returns an error `EAGAIN`, and the close operation continues asynchronously, attempting to send the remaining data.
 - The `SO_LINGER` option may be set to zero to indicate that `close(fd)` should be abortive. A call to `close(fd)` tears down the connection by emitting a reset segment to the remote end (abandoning any data remaining in the socket's send queue) and returns successfully without blocking.
- If `close(fd)` is called on a TCP socket in a pre-established state the file description and socket are simply closed and removed, regardless of how `SO_LINGER` is set, except on Linux platforms where `SYN_RECEIVED` is dealt with as an established state for the purposes of `close(fd)`.
- Calling `close(fd)` on a listening TCP socket closes and removes the socket and aborts each of the connections on the socket's pending and completed connection queues.

15.3.1 Errors

A call to `close()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------|---|
| EAGAIN | The linger timer expired for a lingering <code>close()</code> call and the socket has not yet been successfully closed. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| EINTR | The system was interrupted by a caught signal. |

15.3.2 Common cases

A TCP socket is created and connected to a peer; other socket calls are made, most likely `send()` and `recv()`, but the `SO_LINGER` option is not set. `close()` is then called and the connection is gracefully closed: `socket_1; ...; close_2`

A UDP socket is created and socket calls are made on it, mostly `send()` and `recv()` calls; the socket is then closed: `socket_1; ...; close_10`

15.3.3 API

```
Posix:    int close(int fildes);
FreeBSD:  int close(int d);
Linux:    int close(int fd);
WinXP:    int closesocket(SOCKET s);
```

In the Posix interface:

- `fildes` is the file descriptor to close, corresponding to the `fd` argument of the model `close()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.3.4 Model details

The following errors are not modelled:

- In Posix and on FreeBSD and Linux, `EIO` means an I/O error occurred while reading from or writing to the file system. Since we model only sockets, not file systems, we do not model this error.
- On FreeBSD, `ENOSPC` means the underlying object did not fit, cached data was lost.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.3.5 Summary

| | | |
|----------------------|--------------------------|--|
| <code>close_1</code> | all: fast succeed | Successfully close a file descriptor that is not the last file descriptor for a socket |
| <code>close_2</code> | tcp: fast succeed | Successfully perform a graceful close on the last file descriptor of a synchronised socket |
| <code>close_3</code> | tcp: fast succeed | Successful abortive close of a synchronised socket |

$$\begin{aligned}
&st = \text{SYN_RECEIVED} \wedge \text{linux_arch } h.\text{arch}) \wedge \\
&(sf.t(\text{SO_LINGER}) = \infty \vee \\
&ff.b(\text{O_NONBLOCK}) = \mathbf{T} \wedge sf.t(\text{SO_LINGER}) \neq 0 \wedge \neg \text{linux_arch } h.\text{arch}) \wedge \\
&fd \in \mathbf{dom}(fds) \wedge \\
&fid = fds[fd] \wedge \\
&fid_ref_count(fds, fid) = 1 \wedge \\
&fds' = fds \setminus \{fd\} \wedge \\
&fid \notin (\mathbf{dom}(files))
\end{aligned}$$

Description

A `close(fd)` call is performed on the TCP socket `sid` referenced by file descriptor `fd` which is the only file descriptor referencing the socket's file description: $fid_ref_count(fds, fid) = 1$. The TCP socket `sid` is in a synchronised state, i.e. a state \geq ESTABLISHED, or on Linux it may be in the SYN_RECEIVED state.

In the common case the socket's linger option is not set, $sf.t(\text{SO_LINGER}) = \infty$, and regardless of whether the socket is in non-blocking mode or not, i.e. $ff.b(\text{O_NONBLOCK})$ is unconstrained, the call to `close()` proceeds successfully without blocking.

On all platforms except for Linux, if the socket is in non-blocking mode $ff.b(\text{O_NONBLOCK}) = \mathbf{T}$ the linger option may be set with a positive duration: $sf.t(\text{SO_LINGER}) \neq 0$). In this case the option is ignored giving precedence to the socket's non-blocking semantics. The `close()` call succeeds without blocking.

The `close(fd)` call proceeds by a `tid-close(fd)` transition leaving the host in the successful return state `RET(OK())`. The final socket is marked as unable to send and receive further data, $cantsndmore = \mathbf{T} \wedge cantrcvmore = \mathbf{T}$, eventually causing TCP to transmit all remaining data in the socket's send queue and perform a graceful close.

In the final host state, the mapping of file descriptor `fd` to file descriptor index `fid` is removed from the file descriptors finite map $fds' = fds \setminus \{fd\}$ and the file description entry `fid` is removed from the finite map of file descriptors $files \setminus \{fid\}$. The socket entry itself, $(sid, \text{SOCK}(\uparrow fid, \dots))$ is not destroyed at this point; it remains until the TCP connection has been successfully closed.

Variations

| | |
|-------|---|
| Linux | The socket can be in the SYN_RECEIVED state or in one of the synchronised states \geq ESTABLISHED. On Linux, non-blocking semantics do not take precedence over the SO_LINGER option, i.e. if the socket is non-blocking, $ff.b(\text{O_NONBLOCK}) = \mathbf{T}$ and a linger option is set to a non-zero value, $sf.t(\text{SO_LINGER}) \neq 0$, the socket may block on a call to <code>close()</code> . See also close_4 (p140). |
|-------|---|

`close_3` tcp: fast succeed Successful abortive close of a synchronised socket

$$\begin{array}{ccc}
h \{ts := ts \oplus (tid \mapsto (\text{RUN})_d); & \xrightarrow{\text{tid-close}(fd)} & h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
fds := fds; & & fds := fds'; \\
files := files \oplus & & files := files; \\
\quad [(fid, \text{FILE}(\text{FT_SOCKET}(sid), ff))]; & & socks := socks \oplus [(sid, sock')]; \\
socks := socks \oplus & & oq := oq' \\
\quad [(sid, sock)]; & & \\
oq := oq' & &
\end{array}$$

$$\begin{aligned}
&(st \in \{\text{ESTABLISHED}; \text{FIN_WAIT_1}; \text{CLOSING}; \text{FIN_WAIT_2}; \\
&\quad \text{TIME_WAIT}; \text{CLOSE_WAIT}; \text{LAST_ACK}\} \vee \\
&st = \text{SYN_RECEIVED} \wedge \text{linux_arch } h.\text{arch}) \wedge \\
&sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \\
&\quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge \\
&sf.t(\text{SO_LINGER}) = 0 \wedge \\
&fd \in \mathbf{dom}(fds) \wedge
\end{aligned}$$

$$\begin{aligned}
&fid = fds[fd] \wedge \\
&fid_ref_count(fds, fid) = 1 \wedge \\
&fds' = fds \setminus fd \wedge \\
&fid \notin (\mathbf{dom}(files)) \wedge \\
&sid \notin \mathbf{dom}(socks) \wedge \\
&sock' = (tcp_close \ h.arch \ sock) \langle [fid := *] \rangle \wedge \\
&seg \in \text{make_rst_segment_from_cb } cb(i_1, i_2, p_1, p_2) \wedge \\
&\text{enqueue_and_ignore_fail } h.arch \ h.rttab \ h.ifds[\text{TCP } seg] \ oq \ oq'
\end{aligned}$$

Description

A $\text{close}(fd)$ call is performed on the TCP socket sid referenced by file descriptor fd which is the only file descriptor referencing the socket's file description: $fid_ref_count(fds, fid) = 1$. The TCP socket sid is in a synchronised state, i.e. a state \geq ESTABLISHED, except on Linux platforms where it may be in the SYN_RECEIVED state.

The socket's linger option is set to a duration of zero, $sf.t(\text{SO_LINGER}) = 0$, to signify that an abortive closure of socket sid is required.

The $\text{close}(fd)$ call proceeds by a $tid.\text{close}(fd)$ transition leaving the host in the successful return state $\text{RET}(\text{OK}())$. A reset segment seg is constructed from the socket's control block cb and address quad (i_1, i_2, p_1, p_2) and is appended to the host's output queue, oq , by the function $\text{enqueue_and_ignore_fail}$ (p118), to create new output queue oq' . The $\text{enqueue_and_ignore_fail}$ function always succeeds; if it is not possible to add the reset segment seg to the output queue the corresponding error code is ignored and the reset segment is not queued for transmission.

The mapping of file descriptor fd to index fid is removed from the file descriptors finite map $fds' = fds \setminus fd$ and the file description entry indexed by fid is removed from the finite map of file descriptions. The socket is put in the CLOSED state, shutdown for reading and writing, has its control block reset, and its send and receive queues emptied; this is done by the auxiliary function tcp_close (p121). Additionally, its file description field is cleared.

Variations

| | |
|-------|--|
| Linux | The socket can be in the SYN_RECEIVED state or in one of the synchronised states \geq ESTABLISHED. |
|-------|--|

close_4 **tcp: block** Block on a lingering close on the last file descriptor of a synchronised socket

$$\begin{aligned}
&h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d)]; \\
&fds := fds; \\
&files := files \oplus \\
&\quad [(fid, \text{FILE}(\text{FT_SOCKET}(sid), ff))]; \\
&socks := socks \oplus \\
&\quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \text{cantsndmore}, \text{cantrcvmore}, \\
&\quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))] \\
\hline
&\text{tid.close}(fd) \rightarrow h \langle [ts := ts \oplus (tid \mapsto (\text{CLOSE2}(sid))_{\text{slow_timer}(sf.t(\text{SO_LINGER}))}); \\
&fds := fds'; \\
&files := files; \\
&socks := socks \oplus \\
&\quad [(sid, \text{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \\
&\quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, [], rcvurp, iobc)))] \rangle
\end{aligned}$$

$(st \in \{\text{ESTABLISHED}; \text{FIN_WAIT_1}; \text{CLOSING}; \text{FIN_WAIT_2};$
 $\text{TIME_WAIT}; \text{CLOSE_WAIT}; \text{LAST_ACK}\} \vee$
 $st = \text{SYN_RECEIVED} \wedge \text{linux_arch } h.arch) \wedge$
 $sf.t(\text{SO_LINGER}) \notin \{0; \infty\} \wedge$

$$\begin{aligned}
& (ff.b(O_NONBLOCK) = \mathbf{F} \vee (ff.b(O_NONBLOCK) = \mathbf{T} \wedge \text{linux_arch } h.\text{arch})) \wedge \\
& fd \in \mathbf{dom}(fds) \wedge \\
& fid = fds[fd] \wedge \\
& fid_ref_count(fds, fid) = 1 \wedge \\
& fds' = fds \setminus fd \wedge \\
& fid \notin (\mathbf{dom}(files))
\end{aligned}$$

Description

A $\text{close}(fd)$ call is performed on the TCP socket sid referenced by file descriptor fd which is the only file descriptor referencing the socket's file description: $fid_ref_count(fds, fid) = 1$. The TCP socket sid has a blocking mode of operation, $ff.b(O_NONBLOCK) = \mathbf{F}$, and is in a synchronised state, i.e. a state \geq ESTABLISHED.

On Linux, the socket is also permitted to be in the SYN_RECEIVED state and it may have non-blocking semantics $ff.b(O_NONBLOCK) = \mathbf{T}$, because the linger option takes precedence over non-blocking semantics.

The socket's linger option is set to a positive duration and is neither zero (which signifies an immediate abortive close of the socket) nor infinity (which signifies that the linger option has not been set), $sf.t(\text{SO_LINGER}) \notin \{0; \infty\}$. The close call blocks for a maximum duration that is the linger option duration in seconds, during which time TCP attempts to send all remaining data in the socket's send buffer and gracefully close the connection.

The $\text{close}(fd)$ call proceeds by a $tid.\text{close}(fd)$ transition leaving the host in the blocked state $\text{CLOSE2}(sid)$. The socket is marked as unable to send and receive further data, $cantsndmore = \mathbf{T} \wedge cantrcvmore = \mathbf{T}$; this eventually causes TCP to send all remaining data in the socket's send queue and perform a graceful close.

In the final host state, the mapping of file descriptor fd to file descriptor index fid is removed from the file descriptors finite map $fds' = fds \setminus fd$ and file description entry fid is removed from the finite map of file descriptors. The socket entry itself, $(sid, \text{SOCK}(\uparrow fid, \dots))$, is not destroyed at this point; it remains until the TCP socket has been successfully closed by future asynchronous events.

Variations

| | |
|-------|---|
| Linux | <p>The socket can be in the SYN_RECEIVED state or in one of the synchronised states \geq ESTABLISHED.</p> <p>On Linux, non-blocking semantics do not take precedence over the SO_LINGER option, i.e. if the socket is non-blocking, $ff.b(O_NONBLOCK) = \mathbf{T}$ and a linger option is set to a non-zero value, $sf.t(\text{SO_LINGER}) \neq 0$ the socket may block on a call to $\text{close}()$.</p> |
|-------|---|

close_5 **tcp: slow urgent succeed** Successful completion of a lingering close on a synchronised socket

$$\begin{aligned}
& h \llbracket ts := ts \oplus (tid \mapsto (\text{CLOSE2}(sid))_d); \\
& \text{socks} := \text{socks} \oplus \\
& \quad \llbracket (sid, \text{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, [], sndurp, [], rcvurp, iobc)) \rrbracket \\
\tau \rightarrow & h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
& \text{socks} := \text{socks} \oplus \\
& \quad \llbracket (sid, \text{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, [], sndurp, [], rcvurp, iobc)) \rrbracket
\end{aligned}$$

$st \in \{\text{TIME_WAIT}; \text{CLOSED}; \text{FIN_WAIT_2}\}$

Description

A previous call to `close()` with the `linger` option set on the socket blocked leaving thread `tid` in the `CLOSE2(sid)` state. The socket `sid` has successfully transmitted all the data in its send queue, $sndq = []$, and has completed a graceful close of the connection: $st \in \{\text{TIME_WAIT}; \text{CLOSED}; \text{FIN_WAIT_2}\}$.

The rule proceeds via a τ transition leaving thread `tid` in the `RET(OK())` state to return successfully from the blocked `close()` call. The socket remains in a closed state.

Note that the asynchronous sending of any remaining data in the send queue and graceful closing of the connection is handled by other rules. This rule applies once these events have reached a successful conclusion.

close_6 **tcp: slow nonurgent fail** Fail with **EAGAIN**: unsuccessful completion of a lingering close on a synchronised socket

$$h \langle \{ts := ts \oplus (tid \mapsto (\text{CLOSE2}(sid))_d); \quad \xrightarrow{\tau} \quad h \langle \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched_timer}});$$

$$socks := socks \oplus [(sid, sock)] \quad socks := socks \oplus [(sid, sock)]$$

$$\rangle \quad \rangle$$

$$sock = \text{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},$$

$$\text{TCP_Sock}(st, cb, *, sndq, sndurp, [], rcvurp, iobc) \wedge$$

$$\text{timer_expires } d \wedge$$

$$st \notin \{\text{TIME_WAIT}; \text{CLOSED}\}$$

Description

A previous call to `close()` with the `linger` option set on the socket blocked, leaving thread `tid` in the `CLOSE2(sid)` state. The linger timer has expired, `timer_expires d`, before the socket has been successfully closed: $st \notin \{\text{TIME_WAIT}; \text{CLOSED}\}$.

The rule proceeds via a τ transition leaving thread `tid` in the `RET(FAIL EAGAIN)` state to return error `EAGAIN` from the blocked `close()` call. The socket remains in a synchronised state and is not destroyed until the socket has been successfully closed by future asynchronous events.

The asynchronous transmission of any remaining data in the send queue and the graceful closing of the connection is handled by other rules. This rule is only predicated on the unsuccessfulness of these operations, i.e. $st \notin \{\text{TIME_WAIT}; \text{CLOSED}\}$. When the linger timer expires the socket could be (a) still attempting to successfully transmit the data in the send queue, or (b) be someway through the graceful close operation. The exact state of the socket is not important here, explaining the relatively unconstrained socket state in the rule.

close_7 **tcp: fast succeed** Successfully close the last file descriptor for a socket in the **CLOSED**, **SYN_SENT** or **SYN_RECEIVED** states.

$$h \langle \{ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$fds := fds;$$

$$files := files \oplus [(fid, \text{FILE}(\text{FT_SOCKET}(sid), ff))];$$

$$socks := socks \oplus [(sid, sock)] \rangle$$

$$\xrightarrow{tid \cdot \text{close}(fd)} \quad h \langle \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}});$$

$$fds := fds';$$

$$files := files;$$

$$socks := socks \rangle$$

$$(tcp_sock.st \in \{\text{CLOSED}; \text{SYN_SENT}\} \vee$$

$$tcp_sock.st = \text{SYN_RECEIVED} \wedge \neg \text{linux_arch } h.arch) \wedge$$

$$\text{TCP_PROTO}(tcp_sock) = sock.pr \wedge$$

$$fid \notin (\text{dom}(files)) \wedge$$

$$sid \notin (\text{dom}(socks)) \wedge$$

$$fd \in \text{dom}(fds) \wedge$$

$$fid = fds[fd] \wedge$$

$$\text{fid_ref_count}(fds, fid) = 1 \wedge$$

$$fds' = fds \setminus fd$$

Description

A $\text{close}(fd)$ call is performed on the TCP socket $sock$, identified by sid and referenced by file descriptor fd which is the only file descriptor referencing the socket's file description: $\text{fid_ref_count}(fds, fd) = 1$. The TCP socket $sock$ is not in a synchronised state: $st \in \{\text{CLOSED}; \text{SYN_SENT}\}$.

The $\text{close}(fd)$ call proceeds by a $\text{tid}\text{-close}(fd)$ transition leaving the host in the successful return state $\text{RET}(\text{OK}())$.

The mapping of file descriptor fd to file descriptor index fid is removed from the host's finite map of file descriptors; the file description entry for fd is removed from the host's finite map of file descriptors; and the socket entry $(sid, sock)$ is removed from the host's finite map of sockets.

Variations

| | |
|-------|---|
| Linux | The rule does not apply if the socket is in state SYN_RECEIVED: for the purposes of $\text{close}()$ this is treated as a synchronised state on Linux. Note that the socket $sock$ is not in a synchronised state and thus has no data in its send queue ready for transmission. Closing an unsynchronised socket simply involves deleting the socket entry and removing all references to it. These operations are performed immediately by the rule, hence the socket's SO_LINGER option is not constrained because it has no effect regardless of how it may be set. |
|-------|---|

close_8 **tcp: fast succeed** Successfully close the last file descriptor for a listening TCP socket

$$\begin{array}{l} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ fds := fds; \\ files := files \oplus [(fid, \text{FILE}(\text{FT_SOCKET}(sid), ff))]; \\ socks := socks \oplus [(sid, sock)]; \\ listen := listen; \\ oq := oq \rangle \\ \hline \text{tid}\text{-close}(fd) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\ fds := fds'; \\ files := files; \\ socks := socks'; \\ listen := listen'; \\ oq := oq' \rangle \end{array}$$

$$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$$

$$fd \in \text{dom}(fds) \wedge$$

$$fid = fds[fd] \wedge$$

$$\text{fid_ref_count}(fds, fid) = 1 \wedge$$

$$fid \notin (\text{dom}(files)) \wedge$$

$$sid \notin (\text{dom}(socks)) \wedge$$

(* cantrcvmore/cantsndmore unconstrained under BSD, as may have previously called shutdown *)

(* MS: this is more of an assertion than a condition, so we could get away without it *)

$$(\text{bsd_arch } h.\text{arch} \vee (\text{cantsndmore} = \mathbf{F} \wedge \text{cantrcvmore} = \mathbf{F})) \wedge$$

(* BSD and Linux do not send RSTs to sockets on $lis.q_0$. *)

$$socks_to_rst = \{(sock', tcp_sock') \mid \exists sid'. sid' \in lis.q \wedge$$

$$sock' = socks[sid'] \wedge$$

$$\text{TCP_PROTO}(tcp_sock') = sock'.pr \wedge$$

$$tcp_sock'.st \notin \{\text{CLOSED}; \text{LISTEN}; \text{SYN_SENT}\} \wedge$$

$$socks_to_rst_list \in \text{ORDERINGS } socks_to_rst \wedge$$

$$\text{card } socks_to_rst = \text{length } segs \wedge$$

$$(\text{let } make_rst_seg = \lambda(sock', tcp_sock').$$

$$\text{make_rst_segment_from_cb } tcp_sock'.cb(\text{the } sock'.is_1, \text{the } sock'.is_2, \text{the } sock'.ps_1, \text{the } sock'.ps_2)$$

$$\text{in}$$

$$\text{every } \mathbf{I}(\text{map2}(\lambda s' \text{ seg}' . seg' \in make_rst_seg \ s') socks_to_rst_list \ segs)) \wedge$$

(* Note this is a clear example of where fuzzy timing is needed: should these really all have exactly the same time always? *)

$$\text{enqueue_each_and_ignore_fail } h.arch \ h.rttab \ h.ifds(\text{map } \text{TCP } segs) oq \ oq' \wedge$$

$$fds' = fds \setminus fd \wedge$$

$$listen' = \text{filter}(\lambda sid'.sid' \neq sid)listen \wedge$$

$$socks' = socks|_{\{sid'|sid' \notin lis.q_0 @ lis.q\}}$$

Description

A `close(fd)` call is performed on the TCP socket *sock* referenced by file descriptor *fd* which is the only file descriptor referencing the socket's file description *fid*, $\text{fid_ref_count}(fds, fid) = 1$. Socket *sock* is locally bound to port p_1 and one or more local IP addresses is_1 , and is in the LISTEN state.

The listening socket *sock* may have ESTABLISHED incoming connections on its connection queue $lis.q$ and incomplete incoming connection attempts on queue $lis.q_0$. Each connection, regardless of whether it is complete or not, is represented by a **socket** entry in $h.socks$ and its corresponding index *sid* is on the respective queue. These connections have not been accepted by any thread through a call to `accept()` and are dropped on the closure of socket *sock*.

A set of reset segments *rst*_s*_to_go* is created using the auxiliary function `make_rst_segment_from_cb` (p109) for each of the sockets referenced by both queues. This is performed by looking up each socket *sock'* for every *sid'* in the concatenation of both queues, $lis.q_0 @ lis.q$, and extracting their address quads ($sock'.is_1, sock'.is_2, sock'.ps_1, sock'.ps_2$) and control blocks *cb* for use by `make_rst_segment_from_cb`.

The set of reset segments *rst*_s*_to_go* is constrained to a list, *segs*, and queued by the auxiliary function `enqueue_each_and_ignore_fail` on the hosts output queue $h.oq$. The `enqueue_each_and_ignore_fail` function always succeeds; if it is not possible to add any of the reset segments *segs* to the output queue $h.oq$, the corresponding error codes are ignored and the reset segments in error are ultimately not queued for transmission. This is sensible behaviour as the sockets for these connections are about to be deleted: if a reset segment does not successfully abort the remote end of the connection, perhaps because it could not be transmitted in the first place, any future incoming segments should not match any other socket in the system and will be dropped.

The `close(fd)` call proceeds by a `tid-close(fd)` transition leaving the host in the successful return state `RET(OK())`.

In the final host state, the mapping of file descriptor *fd* to file descriptor index *fid* is removed from the file descriptors finite map $fds' = fds \setminus fd$ and file description entry *fid* is removed from the finite map of file descriptors $h.files$. The socket entry *sock* is removed from the hosts finite map of sockets $h.socks$ and the socket's *sid* value is removed from the host's list of listening sockets $h.listen$ by $listen' = \text{filter}(\lambda sid'.sid' \neq sid)listen$. Finally, all the sockets in $h.socks$ that were referenced on one of the queues $lis.q_0$ and $lis.q$, are removed by $socks' = socks|_{\{sid'|sid' \notin lis.q_0 @ lis.q\}}$ as they were not accepted by any thread before socket *sock* was closed.

Model details

The local IP address option is_1 of the socket *sock* is not constrained in this rule. Instead it is constrained by other rules for `bind()` and `listen()` prior to the socket entering the LISTEN state.

close_10 **udp: fast succeed** Successfully close the last file descriptor of a UDP socket

$$\begin{aligned}
 &h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle; \\
 &fds := fds; \\
 &files := files \oplus [(fid, \text{FILE}(\text{FT_SOCKET}(sid), ff))]; \\
 &socks := socks \oplus \\
 &\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \text{cantsndmore}, \text{cantrcvmore}, \\
 &\quad \quad \text{UDP_PROTO}(udp)))] \rangle
 \end{aligned}$$

$$\begin{aligned}
 \frac{tid \cdot \text{close}(fd)}{} &h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}) \rangle; \\
 &fds := fds'; \\
 &files := files; \\
 &socks := socks \rangle
 \end{aligned}$$

$$\begin{aligned}
 &fd \in \mathbf{dom}(fds) \wedge \\
 &fid = fds[fd] \wedge \\
 &fid_ref_count(fds, fid) = 1 \wedge \\
 &fds' = fds \setminus fd \wedge \\
 &fid \notin (\mathbf{dom}(files)) \wedge \\
 &sid \notin (\mathbf{dom}(socks))
 \end{aligned}$$

Description

Consider a UDP socket *sid*, referenced by *fd*, with a file description record indexed by *fid*. *fd* is the only open file descriptor referring to the file description record indexed by *fid*, $fid_ref_count(fds, fid) = 1$. From thread *tid*, which is in the RUN state, a $\text{close}(fd)$ call is made and succeeds.

A $tid \cdot \text{close}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. The socket *sid* is removed from the host's finite map of sockets $socks \oplus \dots$, the file description record indexed by *fid* is removed from the host's finite map of file descriptions $files \oplus \dots$, and *fd* is removed from the host's finite map of file descriptors $fds' = fds \setminus fd$.

15.4 connect() (TCP and UDP)

`connect : fd * ip * port option → unit`

A call to $\text{connect}(fd, ip, port)$ attempts to connect a TCP socket to a peer, or to set the peer address of a UDP socket. Here *fd* is a file descriptor referring to a socket, *ip* is the peer IP address to connect to, and *port* is the peer port.

If *fd* refers to a TCP socket then TCP's connection establishment protocol, often called the *three-way handshake*, will be used to connect the socket to the peer specified by (*ip*, *port*). A peer port must be specified: *port* cannot be set to *. There must be a listening TCP socket at the peer address, otherwise the connection attempt will fail with an `ECONNRESET` or `ECONNREFUSED` error. The local socket must be in the `CLOSED` state: attempts to $\text{connect}()$ to a peer when already synchronised with another peer will fail. To start the connection establishment attempt, a *SYN* segment will be constructed, specifying the initial sequence number and window size for the connection, and possibly the maximum segment size, window scaling, and timestamping. The segment is then enqueued on the host's out-queue; if this fails then the $\text{connect}()$ call fails, otherwise connection establishment proceeds.

If the socket is a blocking one (the `O_NONBLOCK` flag for *fd* is not set), then the call will block until the connection is established, or a timeout expires in which case the error `ETIMEDOUT` is returned.

If the socket is non-blocking (the `O_NONBLOCK` flag is set for *fd*), then the $\text{connect}()$ call will fail with an `EINPROGRESS` error (or `EALREADY` on WinXP), and connection establishment will proceed asynchronously.

Calling $\text{connect}()$ again will indicate the current status of the connection establishment in the returned error: it will fail with `EALREADY` if the connection has not been established, `EISCONN` once the connection has been established, or if the connection establishment failed, an error describing why. Alternatively, $\text{pselect}([], [fd], [], *, -)$ can be used; it will return when *fd* is ready for writing which will be when connection establishment is complete, either successfully or not. On Linux, unsetting the `O_NONBLOCK` flag for *fd* and

then calling `connect()` will block until the connection is established or fails; for WinXP the call will fail with `EALREADY` and the connection establishment will be performed asynchronously still; for FreeBSD the call will fail with `EISCONN` even if the connection has not been established.

Upon completion of connection establishment the socket will be in state `ESTABLISHED`, ready to send and receive data, or `CLOSE_WAIT` if it received a `FIN` segment during connection establishment.

On FreeBSD, if connection establishment fails having sent a `SYN` then further connection establishment attempts are not allowed; on Linux and WinXP further attempts are possible.

If `fd` refers to a UDP socket then the peer address of the socket is set, but no connection is made. The peer address is then the default destination address for subsequent `send()` calls (and the only possible destination address on FreeBSD), and only datagrams with this source address will be delivered to the socket. On FreeBSD the peer port must be specified: a call to `connect(fd, ip, *)` will fail with an `EADDRNOTAVAIL` error; on Linux and WinXP such a call succeeds: datagrams from any port on the host with IP address `ip` will be delivered to the socket. Calling `connect()` on a UDP socket that already has a peer address set is allowed: the peer address will be replaced with the one specified in the call. On FreeBSD if the socket has a pending error, that may be returned when the call is made, and the peer address will also be set.

In order for a socket to connect to a peer or have its peer address set, it must be bound to a local IP and port. If it is not bound to a local port when the `connect()` call is made, then it will be autobound: an unused port for the socket's protocol in the host's ephemeral port range is selected and assigned to the socket. If the socket does not have its local IP address set then it will be bound to the primary IP address of an interface which has a route to the peer. If the socket does have a local IP address set then the interface that this IP address will be the one used to connect to the peer; if this interface does not have a route to the peer then for a TCP socket the `connect()` call will fail when the `SYN` is enqueued on the host's outqueue; for a UDP socket the call will fail on FreeBSD, whereas on Linux and WinXP the `connect()` call will succeed but later `send()` calls to the peer will fail.

For a TCP socket, its binding quad must be unique: there can be no other socket in the host's finite map of sockets with the same binding quad. If the `connect()` call would result in two sockets having the same binding quad then it will fail with an `EADDRINUSE` error. For UDP sockets the same is true on FreeBSD, but on Linux and WinXP multiple sockets may have the same address quad. The socket that matching datagrams are delivered to is architecture-dependent: see *lookup* (p??).

15.4.1 Errors

A call to `connect()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------------------------|--|
| <code>EADDRNOTAVAIL</code> | There is no route to the peer; a port must be specified (<code>port ≠ *</code>); or there are no ephemeral ports left. |
| <code>EADDRINUSE</code> | The address quad that would result if the connection was successful is in use by another socket of the same protocol. |
| <code>EAGAIN</code> | On WinXP, the socket is non-blocking and the connection cannot be established immediately: it will be established asynchronously. [TCP ONLY] |
| <code>EALREADY</code> | A connection attempt is already in progress on the socket but not yet complete: it is in state <code>SYN_SENT</code> or <code>SYN_RECEIVED</code> . [TCP ONLY] |
| <code>ECONNREFUSED</code> | Connection rejected by peer. [TCP ONLY] |
| <code>ECONNRESET</code> | Connection rejected by peer. [TCP ONLY] |
| <code>EHOSTUNREACH</code> | No route to the peer. |
| <code>EINPROGRESS</code> | The socket is non-blocking and the connection cannot be established immediately: it will be established asynchronously. [TCP ONLY] |
| <code>EINVAL</code> | On WinXP, socket is listening. [TCP ONLY] |

| | |
|-------------|---|
| EISCONN | Socket already connected. [TCP ONLY] |
| ENETDOWN | The interface used to reach the peer is down. |
| ENETUNREACH | No route to the peer. |
| EOPNOTSUPP | On FreeBSD, socket is listening. [TCP ONLY] |
| ETIMEDOUT | The connection attempt timed out before a connection was established for a socket. [TCP ONLY] |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| EINTR | The system was interrupted by a caught signal. |
| ENOBUFS | Out of resources. |

15.4.2 Common cases

TCP: *socket_1; connect_1; ...*

UDP: *socket_1; bind_1; connect_8; ...*

15.4.3 API

```
Posix:    int connect(int socket, const struct sockaddr *address, socklen_t address_len);
FreeBSD:  int connect(int s, const struct sockaddr *name, socklen_t namelen);
Linux:    int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
WinXP:    int connect(SOCKET s, const struct sockaddr* name, int namelen);
```

In the Posix interface:

- **socket** is a file descriptor referring to the socket to make a connection on, corresponding to the **fd** argument of the model `connect()`.
- **address** is a pointer to a `sockaddr` structure of length **address_len** specifying the peer to connect to. `sockaddr` is a generic socket address structure: what is used for the model `connect()` is an internet socket address structure `sockaddr_in`. The `sin_family` member is set to `AF_INET`; the `sin_port` is the port to connect to, corresponding to the `port` argument of the model `connect()`: `sin_port = 0` corresponds to `port = *` and `sin_port=p` corresponds to `port = ↑ p`; the `sin_addr.s_addr` member of the structure corresponds to the `ip` argument of the model `connect()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

Note: For UDP sockets, the Winsock Reference says "The default destination can be changed by simply calling `connect` again, even if the socket is already connected. Any datagrams queued for receipt are discarded if name is different from the previous `connect`." This is not the case.

15.4.4 Model details

If the call blocks then the thread enters state `CONNECT2(sid)` where `sid` is the identifier of the socket attempting to establish a connection.

The following errors are not modelled:

- **EAFNOSUPPORT** means that the specified address is not a valid address for the address family of the specified socket. The model `connect()` only models the `AF_INET` family of addresses so this error cannot occur.
- **EFAULT** signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to `connect()` that is excluded by the clean interface used in the model.
- **WSAEINPROGRESS** is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- **EINVAL** is a Posix-specific error signifying that the `address_len` argument is not a valid length for the socket's address family or invalid address family in the `sockaddr` structure. The length of the address to connect to is implicit in the model `connect()`, and only the `AF_INET` family of addresses is modelled so this error cannot occur.
- **EPROTOTYPE** is a Posix-specific error meaning that the specified address has a different type than the socket bound to the specified peer address. This error does not occur in any of the implementations as TCP and UDP sockets are dealt with separately.
- **EACCES**, **ELOOP**, and **ENAMETOOLONG** are errors dealing with Unix domain sockets which are not modelled here.

15.4.5 Summary

| | | |
|-------------------|---------------------------------|--|
| <i>connect_1</i> | tcp: rc | Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue |
| <i>connect_2</i> | tcp: slow urgent succeed | Successfully return from blocking state after connection is successfully established |
| <i>connect_3</i> | tcp: slow urgent fail | Fail with the pending error on a socket in the CLOSED state |
| <i>connect_4</i> | tcp: slow urgent fail | Fail: socket has pending error |
| <i>connect_4a</i> | tcp: fast fail | Fail with pending error |
| <i>connect_5</i> | tcp: fast fail | Fail with <code>EALREADY</code> , <code>EINVAL</code> , <code>EISCONN</code> , <code>EOPNOTSUPP</code> : socket already in use |
| <i>connect_5a</i> | all: fast fail | Fail: no route to host |
| <i>connect_5b</i> | all: fast fail | Fail with <code>EADDRINUSE</code> : address already in use |
| <i>connect_5c</i> | all: fast fail | Fail with <code>EADDRNOTAVAIL</code> : no ephemeral ports left |
| <i>connect_5d</i> | tcp: block | Block, entering state <code>CONNECT2</code> : connection attempt already in progress and <code>connect</code> called with blocking semantics |
| <i>connect_6</i> | tcp: fast fail | Fail with <code>EINVAL</code> : socket has been shutdown for writing |
| <i>connect_7</i> | udp: fast succeed | Set peer address on socket with binding quad <code>*, ps₁, *, *</code> |
| <i>connect_8</i> | udp: fast succeed | Set peer address on socket with local address set |
| <i>connect_9</i> | udp: fast fail | Fail with <code>EADDRNOTAVAIL</code> : port must be specified in <code>connect()</code> call on FreeBSD |
| <i>connect_10</i> | udp: fast fail | Fail with pending error on FreeBSD, but still set peer address |

15.4.6 Rules

| |
|---|
| <p><i>connect_1</i> tcp: rc Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue</p> |
|---|

$$h \xrightarrow{tid \cdot \text{connect}(fd, i_2, \uparrow p_2)} h'$$

(* Thread tid is in state RUN and TCP socket sid has binding quad (is_1, ps_1, is_2, ps_2) . *)
 $h = h_0 \llbracket ts := ts_+ \oplus (tid \mapsto (\text{RUN})_d);$
 $socks := socks \oplus$
 $\llbracket (sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
 $\text{TCP_Sock}(st, cb, *, [], *, [], *, \text{NO_OOBDATA}))) \rrbracket;$
 $oq := oq \rrbracket \wedge$

(* Thread tid ends in state t' with updated host sockets and output queue *)
 $h' = h_0 \llbracket ts := ts_+ \oplus (tid \mapsto t');$
 $socks := socks \oplus$
 $\llbracket (sid, \text{SOCK}(\uparrow fid, sf, \uparrow i'_1, \uparrow p'_1, is'_2, ps'_2, es'', \mathbf{F}, \mathbf{F},$
 $\text{TCP_Sock}(st', cb''', *, [], *, [], *, \text{NO_OOBDATA}))) \rrbracket;$
 $bound := bound;$
 $oq := oq' \rrbracket \wedge$

(* File descriptor fd refers to TCP socket sid *)
 $fd \in \mathbf{dom}(h_0.fds) \wedge$
 $fd = h_0.fds[fd] \wedge$
 $h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$

(* Either sid is bound to a local IP address or one of the host's interface has a route to i_2 and i'_1 is one of its IP addresses. If it is not routable, then we will fail below, when we try to enqueue the segment. *)

$i'_1 \in \text{auto_outroute}(i_2, is_1, h.rttab, h.ifds) \wedge$
 (* Notice that auto_outroute never fails if $is_1 \neq *$ (i.e., is specified in the socket). *)

(* The socket is either bound to a local port p'_1 or can be autobound to an ephemeral port p'_1 *)
 $p'_1 \in \text{autobind}(ps_1, \text{PROTO_TCP}, h.socks) \wedge$
 (* If autobinding occurs then sid is added to the head of the host's list of bound sockets. *)
 (**if** $ps_1 = *$ **then** $bound = sid :: h.bound$ **else** $bound = h.bound$) \wedge

(* The socket can be in one of two states: (1) it is in state CLOSED in which case its peer address is not set; it has no pending error; it is not shutdown for writing; and it is not shutdown for reading on non-FreeBSD architectures. Otherwise, (2) on FreeBSD the socket is in state TIME_WAIT, and either is_2 and ps_2 are both set or both are not set. The fact that BSD allows a TIME_WAIT socket to be reconnected means that some fields may contain old data, so we leave them unconstrained here. This is particularly important in the cb . *)

$((st = \text{CLOSED} \wedge is_2 = * \wedge ps_2 = * \wedge$
 $es = * \wedge cantsndmore = \mathbf{F} \wedge (cantrcvmore = \mathbf{F} \vee \text{bsd_arch } h.arch)) \vee$
 $(\text{bsd_arch } h.arch \wedge st = \text{TIME_WAIT} \wedge$
 $(is_2 \neq * \implies ps_2 \neq *) \wedge$
 $(ps_2 \neq * \implies is_2 \neq *))) \wedge$

(* No other TCP sockets on the host have the address quad $(\uparrow i'_1, \uparrow p'_1, \uparrow i_2, \uparrow p_2)$. *)
 $\neg(\exists(sid', s) :: (h.socks \setminus sid).$
 $s.is_1 = \uparrow i'_1 \wedge s.ps_1 = \uparrow p'_1 \wedge$
 $s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge$
 $\text{proto_of } s.pr = \text{PROTO_TCP}) \wedge$

(* Pick an initial sequence number non-deterministically. This allows accidental spoofing of our own connections, but it is unclear how a tighter specification should be expressed. *)
 $iss \in \{n \mid \mathbf{T}\} \wedge$

(* If windows-scaling is to be requested for the connection then $request_r_scale = \uparrow n$ where n is a valid window scale; otherwise, $request_r_scale = *$. rcv_wnd0 is a valid receive window size. If window scaling is to be requested then the socket's receive window is set to rcv_wnd0 scaled by the window scale factor n ; otherwise it is set to rcv_wnd0 . The socket's receive window is not greater than the size of the socket's receive buffer. We must allow implementations to either (a) not implement window scaling, or (b) choose on a per-connection basis whether to do window scaling or not. This permits both. *)

```
(request_r_scale : num option) ∈ {*} ∪ {↑ n | n ≥ 0 ∧ n ≤ TCP_MAXWINSIZE} ∧
(rcv_wnd0 : num) ∈ {n | n > 0 ∧ n ≤ TCP_MAXWIN} ∧
(rcv_wnd : num) = rcv_wnd0 << (option_case 0 I request_r_scale) ∧
rcv_wnd ≤ sf.n(SO_RCVBUF) ∧
```

(* Either advertise a maximum segment size, $advms$, that is between 1 and 65535 - 40, or advertise no maximum segment size. If one is advertised, $advms' = \uparrow advms$; otherwise, $advms' = *$. *)

```
advms ∈ {n | n ≥ 1 ∧ n ≤ (65535 - 40)} ∧
advms' ∈ {*, ↑ advms} ∧
```

(* If time-stamping is to be requested for the connection, then $tf_req_tstamp' = \mathbf{T}$; otherwise $tf_req_tstamp' = \mathbf{F}$. *)

```
tf_req_tstamp' ∈ {F, T} ∧ (* do timestamp? *)
```

(* If there is no segment currently being timed for this socket (the expected case) then the SYN segment will be timed, with t_rttseg' set to the current time and the initial sequence number for the connection, iss . *)

```
(let t_rttseg' = if IS_NONE cb.t_rttseg then
    ↑(ticks_of h.ticks, iss)
  else
    cb.t_rttseg in
```

(* Update the socket's control block to cb' , which is cb except we: (1) start the retransmit and connection establishment timers; (2) set the snd_una , snd_nxt , snd_max , iss fields based on the initial sequence number chosen; (3) set the rcv_wnd , rcv_adv , and $tf_rxwin0sent$ fields based on the receive window chosen; (4) record whether or not to do windows scaling, time-stamping, and what the advertised maximum segment size is; and (5) store the segment to time. *)

```
cb' = cb { tt_rexmt := start_tt_rexmtsyn h.arch 0 F cb.t_rttinf;
  tt_conn_est := ↑((( ))_slow_timer TCPTV_KEEP_INIT);
```

```
snd_una := iss;
snd_nxt := iss + 1;
snd_max := iss + 1;
iss := iss;
rcv_wnd := rcv_wnd;
rcv_adv := cb.rcv_nxt + rcv_wnd;
```

(* since rcv_nxt is 0 at this point (since we do not yet know), this is a bit odd. But it models BSD behaviour. *)

```
tf_rxwin0sent := (rcv_wnd = 0);
request_r_scale := request_r_scale; (* store whether we requested WS and if so what *)
t_maxseg := cb.t_maxseg; (* do not change this *)
t_advms := advms'; (* store what mss we advertised; * or ↑ v *)
tf_req_tstamp := tf_req_tstamp';
last_ack_sent := tcp_seq_foreign 0w;
t_rttseg := t_rttseg'
```

⌋) ∧

(* now build the segment (using an auxiliary, since we might have to retransmit it) *)

(* Make a SYN segment based on the updated control block and the socket's address quad; see make_syn_segment (p106) for details. *)

choose $seg :: \text{make_syn_segment } cb'(i'_1, i_2, p'_1, p_2)(\text{ticks_of } h.\text{ticks}).$

(* and send it out... *)

(* If possible, enqueue the segment seg on the host's outqueue. The auxiliary function `rollback_tcp_output` (p117) is used for this; if the segment is a well-formed segment, there is a route to the peer from i'_1 , and there are no buffer allocation failures, $outsegs' \neq []$, then the segment is enqueued on the host's outqueue, oq , resulting in a new outqueue, oq' . The socket's control block is left as cb' which is described above. Otherwise an error may have occurred; possible errors are: (1) ENOBUFS indicating a buffer allocation failure; (2) a routing error; or (3) EADDRNOTAVAIL on FreeBSD or EINVAL on Linux indicating that the segment would cause a loopback packet to appear on the wire (on WINXP the segment is silently dropped with no error in this case). If an error does occur then the socket's control block reverts to cb , the control block when the call was made. *)

∃ $outsegs'$.

`rollback_tcp_output` $\mathbf{F}(\text{TCP } seg)h.arch h.rttab h.ifds \mathbf{T}$
 $(cb \{ \text{snd_nxt} := iss;$
 $\text{snd_max} := iss;$
 $tt_delack := *;$
 $last_ack_sent := \text{tcp_seq_foreign } 0w;$
 $rcv_adv := \text{tcp_seq_foreign } 0w$
 $\})cb'(cb'', es', outsegs') \wedge$

$cb''' = (\text{if } (outsegs' \neq [] \vee \text{windows_arch } h.arch) \text{ then } cb'' \text{ else } cb) \wedge$
 $\text{enqueue_oq_list_qinfo}(oq, outsegs', oq') \wedge$

(* If the socket is a blocking one, its O_NONBLOCK flag is not set, then the call will block, entering state `CONNECT2(sid)` and leaving the socket in state `SYN_SENT` with peer address $(\uparrow i_2, \uparrow p_2)$ and, if the segment could not be enqueued, its pending error set to the error resulting from the attempt to enqueue the segment.

If the socket is non-blocking, its O_NONBLOCK flag is set, and the segment was enqueued on the host's outqueue, then the call will fail with an `EINPROGRESS` error (or `EAGAIN` on WinXP). The socket will be left in state `SYN_SENT` with peer address $(\uparrow i_2, \uparrow p_2)$. Otherwise, if the segment was not enqueued, then the call will fail with the error resulting from attempting to enqueue it, $\uparrow err$; the socket will be left in state `CLOSED` with no peer address set. *)

(* In the case of BSD, if we connect via the loopback interface, then the segment exchange occurs so fast that the socket has connected before the connect-calling thread regains control. When it does, it sees that the socket has been connected, and therefore returns with success rather than `EINPROGRESS`. Since this behaviour is due to timing, however, it may be possible for the connect call to return before all the segments have been sent, for example if there was an artificially imposed delay on the loopback interface. This behaviour is therefore made nondeterministic, for a BSD non-blocking socket connecting via loopback, in that it may either fail immediately, or be blocked for a short time. Linux does not exhibit this behaviour.*)

((* blocking socket, or BSD and using loopback interface *)

$((\neg ff.b(\text{O_NONBLOCK}) \vee (\text{bsd_arch } h.arch \wedge i_2 \in \text{local_lips } h.ifds)) \wedge$

$t' = (\text{CONNECT2}(sid))_{\text{never_timer}} \wedge rc = \text{BLOCK} \wedge$

$es'' = es' \wedge st' = \text{SYN_SENT} \wedge is'_2 = \uparrow i_2 \wedge ps'_2 = \uparrow p_2) \vee$

(* non-blocking socket *)

$(ff.b(\text{O_NONBLOCK}) \wedge$

$es = * \wedge$

$(err = (\text{if } \text{windows_arch } h.arch \text{ then } \text{EAGAIN} \text{ else } \text{EINPROGRESS})) \vee \uparrow err = es') \wedge$

$t' = (\text{RET}(\text{FAIL } err))_{\text{sched_timer}} \wedge rc = \text{FAST_FAIL} \wedge es'' = * \wedge$

if $oq = oq'$ **then**

$st' = \text{CLOSED} \wedge is'_2 = * \wedge ps'_2 = *$

else

$$st' = \text{SYN_SENT} \wedge is'_2 = \uparrow i_2 \wedge ps'_2 = \uparrow p_2)$$

Description

From thread tid , a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made where fd refers to a TCP socket. The socket is in state CLOSED with no peer address set, no pending error, and not shutdown for reading or writing. A SYN segment is created to being connection establishment, and is enqueued on the host's out-queue.

If the socket is a blocking one (its O_NONBLOCK flag is not set) then the call will block: a $tid\text{-connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state CONNECT2(sid). If the socket is non-blocking (its O_NONBLOCK flag is set) and the segment enqueueing was successful then the call will fail: a $tid\text{-connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state RET(FAIL EINPROGRESS) (or RET(FAIL EAGAIN) on WinXP); connection establishment will proceed asynchronously. Otherwise, if the enqueueing did not succeed, the call will fail with an error err : a $tid\text{-connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread in state RET(FAIL err).

For further details see the in-line comments above.

Variations

| | |
|---------|--|
| FreeBSD | The socket may also be in state TIME_WAIT when the $\text{connect}()$ call is made, with either both its peer IP and port set, or neither set. The socket may be shutdown for reading when the $\text{connect}()$ call is made. |
| WinXP | If there is an early buffer allocation failure when enqueueing the segment, then it will not be placed on the host's out-queue and $es' = \text{ENOBUFS}$; the socket's control block will be cb' with its snd_next and snd_max fields set to the initial sequence number, its $last_ack_seen$ and rcv_adv fields set to 0, its tt_delack option set to *, its tt_rearmt timer stopped, and its $tf_rxwin0sent$ and t_rttseg fields reset. If there is no route from an interface specified by the local IP address i_1 to the foreign IP address i_2 then the socket's control block will be cb' with its snd_next field set to the initial sequence number, its $last_ack_sent$ and rcv_adv fields set to 0, and its tt_delack option set to *. If the segment would cause a loopback packet to be sent on the wire then the socket's control block will be cb' . |

connect_2 tcp: slow urgent succeed Successfully return from blocking state after connection is successfully established

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{CONNECT2 } sid)_d) \rangle \rangle \xrightarrow{\tau} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}) \rangle \rangle$$

$$\begin{aligned} \text{TCP_PROTO}(tcp_sock) &= (h.socks[sid]).pr \wedge \\ tcp_sock.st &\in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge \\ (\neg \exists tid' d'. (tid' \in \text{dom}(ts)) \wedge (tid' \neq tid) \wedge \\ &ts[tid'] = (\text{CONNECT2 } sid)_{d'}) \end{aligned}$$

Description

Thread tid is blocked in state CONNECT2(sid) where sid identifies a TCP socket which is in state ESTABLISHED: the connection establishment has been successfully completed; or CLOSE_WAIT: connection establishment successfully completed but a FIN was received during establishment. tid is the only thread which is blocked waiting for the socket sid to establish a connection. As connection establishment has now completed, the thread can successfully return from the blocked state.

A τ transition is made, leaving the thread state RET(OK()).

connect_3 **tcp: slow urgent fail** Fail with the pending error on a socket in the CLOSED state

$$h \langle ts := ts \oplus (tid \mapsto (\text{CONNECT2 } sid)_d); \quad \xrightarrow{\tau} \quad h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}});$$

$$socks := socks \oplus \quad socks := socks \oplus$$

$$[(sid, sock \langle es := \uparrow e \rangle)] \quad [(sid, sock \langle es := * \rangle)]$$

TCP_PROTO(*tcp_sock*) = *sock.pr* \wedge
tcp_sock.st = CLOSED \wedge
 (bsd_arch *h.arch* \implies *tcp_sock.cb.bsd_cantconnect* = **T**)

Description

Thread *tid* is blocked in the CONNECT2(*sid*) state where *sid* identifies a TCP socket *sock* that is in the CLOSED state: connection establishment has failed, leaving the socket in a pending error state $\uparrow e$. Usually this occurs when there is no listening TCP socket at the peer address, giving an error of ECONNREFUSED or ECONNRESET; or when the connection establishment timer expired, giving an error of ETIMEDOUT. The call now returns, failing with the error *e*, and clearing the pending error field of the socket.

A τ transition is made, leaving the thread state RET(FAIL *e*).

Variations

| | |
|---------|---|
| FreeBSD | When connection establishment failed, the <i>bsd_cantconnect</i> flag in the control block would have been set, the socket's <i>cantsndmore</i> and <i>cantrcvmore</i> flags would have been set and its local address binding would have been removed. This renders the sockets useless: call to <i>bind()</i> , <i>connect()</i> , and <i>listen()</i> will all fail. |
|---------|---|

connect_4 **tcp: slow urgent fail** Fail: socket has pending error

$$h \langle ts := ts \oplus (tid \mapsto (\text{CONNECT2 } sid)_d); \quad \xrightarrow{\tau} \quad h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}});$$

$$socks := socks \oplus \quad socks := socks \oplus$$

$$[(sid, sock)] \quad [(sid, sock')]$$

sock = SOCK($\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, \uparrow err, \mathbf{F}, \mathbf{F}$,
 TCP_Sock(SYN_SENT, *cb*, *, [], *, [], *, NO_OOBDATA)) \wedge

(* On WinXP if the error is from routing to an unavailable address, the error is not returned and the socket is left alone. The rexmtsyn timer will retry the SYN transmission and eventually fail. *)

$\neg(\text{windows_arch } h.arch \wedge err = \text{EINVAL}) \wedge$

(if bsd_arch *h.arch* then

(if (*err* = EADDRNOTAVAIL) then

sock' = SOCK($\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \mathbf{F}$,
 TCP_Sock(SYN_SENT, *cb*, *, [], *, [], *, NO_OOBDATA))

else

sock' = SOCK($\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F}$,
 TCP_Sock(CLOSED, initial_cb, *, [], *, [], *, NO_OOBDATA)))

else

(* close the socket, but do not shutdown for reading/writing *)

sock' = SOCK($\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F}$,
 TCP_Sock(CLOSED, *cb'*, *, [], *, [], *, NO_OOBDATA)) \wedge

cb' = initial_cb

)

Description

Thread tid is blocked in the `CONNECT2(sid)` state waiting for a connection to be established. sid identifies a TCP socket $sock$ that has not been shutdown for reading or writing, and has binding quad $(\uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2)$ and pending error err . The socket is in state `SYN_SENT`, is not listening, has empty send and receive queues, and no urgent marks set. The call fails, returning the pending error.

A τ transition is made, leaving the thread state `RET(FAIL err)`. The socket is left in state `CLOSED` with its peer address not set, its pending error cleared, and its control block reset to the initial control block, `initial_cb`.

Variations

| | |
|---------|--|
| FreeBSD | If the pending error is <code>EADDRNOTAVAIL</code> then the error is cleared and returned but the rest of the socket stays the same: it is in state <code>SYN_SENT</code> so the <code>SYN</code> will be retransmitted until it times out. If the pending error is not <code>EADDRNOTAVAIL</code> then the socket is reset as above except that the the socket's local ip and port are cleared |
| WinXP | If the error is <code>EINVAL</code> then this rule does not apply. |

connect_4a **tcp: fast fail** Fail with pending error

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d); \quad \frac{tid \cdot \text{connect}(fd, i_2, \uparrow p_2)}{socks := socks \oplus [(sid, sock \llbracket es := \uparrow err \rrbracket)]} \quad h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}); \quad socks := socks \oplus [(sid, sock \llbracket es := * \rrbracket)] \rrbracket$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &\text{TCP_PROTO}(tcp_sock) = sock.pr \wedge \\ &tcp_sock.st \in \{\text{CLOSED}\} \end{aligned}$$

Description

From thread tid , which is in the `RUN` state, a `connect($fd, i_2, \uparrow p_2$)` call is made. fd refers to a TCP socket $sock$, identified by sid , with pending error err and in state `CLOSED`. The call fails with the pending error.

A `tid.connect($fd, ip, port$)` transition is made, leaving the thread state `RET(FAIL err)` and the socket's pending error clear.

The most likely cause of this behaviour is for a non-blocking `connect($fd, -, -$)` call to have previously been made. The call fails, setting the pending error on the socket, and when `connect()` is called to check the status of connection establishment the error is returned. In such a case err is most likely to be `ECONNREFUSED`, `ECONNRESET`, or `ETIMEDOUT`.

connect_5 **tcp: fast fail** Fail with `EALREADY`, `EINVAL`, `EISCONN`, `EOPNOTSUPP`: socket already in use

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \quad \frac{tid \cdot \text{connect}(fd, i_2, \uparrow p_2)}{h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}) \rrbracket}$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge \\ &\mathbf{case } tcp_sock.st \mathbf{ of} \end{aligned}$$

`SYN_SENT` \rightarrow **if** `ff.b(O_NONBLOCK) = T` **then** $err = \text{EALREADY}$ (* connection already in progress *)

```

    else if windows_arch h.arch then err = EALREADY (* connection already in
                                                progress *)
    else if bsd_arch h.arch then err = EISCONN (* connection being established *)
    else ASSERTION_FAILURE“connect_5:1” || (* never happen *)
SYN_RECEIVED → if ff.b(O_NONBLOCK) = T then err = EALREADY (* connection already in
                                                progress *)

    else if windows_arch h.arch then err = EALREADY
    else if bsd_arch h.arch then err = EISCONN (* connection being established *)
    else ASSERTION_FAILURE“connect_5:2” || (* never happen *)
LISTEN → if windows_arch h.arch then err = EINVAL (* socket is listening *)
    else if bsd_arch h.arch then err = EOPNOTSUPP
    else if linux_arch h.arch then err = EISCONN
    else ASSERTION_FAILURE“connect_5:3” || (* never happen *)
ESTABLISHED → err = EISCONN || (* socket already connected *)
FIN_WAIT_1 → err = EISCONN || (* socket already connected *)
FIN_WAIT_2 → err = EISCONN || (* socket already connected *)
CLOSING → err = EISCONN || (* socket already connected *)
CLOSE_WAIT → err = EISCONN || (* socket already connected *)
LAST_ACK → err = EISCONN || (* socket already connected; seems that fd is valid in this state *)
TIME_WAIT → (windows_arch h.arch ∨ linux_arch h.arch) ∧ err = EISCONN ||
    (* BSD allows a TIME_WAIT socket to be reconnected *)
CLOSED → err = EINVAL ∧ bsd_arch h.arch ∧ tcp_sock.cb.bsd_cantconnect = T

```

Description

From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made where fd refers to a TCP socket identified by sid . The call fails with an error err : if the socket is in state SYN_SENT or SYN_RECEIVED and the socket is non-blocking or the host is a WinXP architecture then $err = \text{EALREADY}$ (EISCONN on FreeBSD); if it is in state LISTEN then on WinXP $err = \text{EINVAL}$, on FreeBSD $err = \text{EOPNOTSUPP}$, and on Linux $err = \text{EISCONN}$; if it is in state ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSING, CLOSE_WAIT, or TIME_WAIT on Linux and WinXP, $err = \text{EISCONN}$; if it is in state CLOSED on FreeBSD and has its $bsd_cantconnect$ flag set then $err = \text{EINVAL}$.

A $tid.\text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } err)$.

Variations

| | |
|---------|--|
| FreeBSD | If the socket is in state TIME_WAIT then the call does not fail: the socket may be reconnected by connect_1 (p148). |
|---------|--|

connect_5a **all: fast fail** Fail: no route to host

$$\begin{aligned}
 & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 & \text{socks} := \text{socks} \oplus \\
 & \quad [(sid, \text{sock} \langle is_1 := *; ps_1 := ps_1 \rangle)] \rangle \\
 \text{tid}.\text{connect}(fd, i_2, \uparrow p_2) \rightarrow & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}); \\
 & \text{socks} := \text{socks} \oplus \\
 & \quad [(sid, \text{sock} \langle is'_1 := is'_1; ps_1 := ps'_1 \rangle)]; \\
 & \text{bound} := \text{bound} \rangle
 \end{aligned}$$

$fd \in \text{dom}(h.fds) \wedge$

$fid = h.fds[fd] \wedge$

$h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$

(if bsd_arch h.arch ∧ proto_of sock.pr = PROTO_TCP then

$is'_1 = \uparrow i'_1 \wedge i'_1 \in \text{local_primary_ips } h.ifds \wedge$

$ps'_1 = \uparrow p'_1 \wedge p'_1 \in \text{autobind}(ps_1, \text{PROTO_TCP}, h.socks) \wedge$

```

    (if ps1 = * then bound = sid :: h.bound else bound = h.bound)
  else is'1 = * ∧ ps'1 = ps1 ∧ bound = h.bound) ∧
case test_outroute_ip(i2, h.rttab, h.ifds, h.arch) of
  ↑ e → err = e
  || _other29 → F ∧
(proto_of sock.pr = PROTO_UDP ⇒ ¬bsd_arch h.arch)

```

Description

From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made. fd refers to a socket identified by sid which does not have a local IP address set. The test_outroute_ip (p82) function is used to check if there is a route from the host to i_2 . There is no route so the call will fail with a routing error err . If there is no interface with a route to the host then on Linux the call fails with ENETUNREACH and on FreeBSD and WinXP it fails with EHOSTUNREACH. If there are interfaces with a route to the host but none of these are up then the call fails with ENETDOWN.

A $tid.\text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } err)$, where err is one of the above errors.

Variations

| | |
|---------|--|
| FreeBSD | This rule does not apply to UDP sockets on FreeBSD. Additionally, if the socket is not bound to a local port then it will be autobound to one and sid will be appended to the head of the host's list of bound sockets, $bound$. The socket's local IP address may be set to $\uparrow i_1$ even though there is no route from i_1 to i_2 . |
|---------|--|

connect_5b **all: fast fail** Fail with EADDRINUSE: address already in use

```

h (ts := ts ⊕ (tid ↦ (RUN)d);
socks := socks ⊕
  [(sid, sock)];
bound := bound)
tid.connect(fd, i2, ↑ p2) → h (ts := ts ⊕ (tid ↦ (RET(FAIL EADDRINUSE))sched_timer);
socks := socks ⊕
  [(sid, sock (is1 := is'1; ps1 := ↑ p'1; is2 := is'2; ps2 := ps'2)]);
bound := bound')

```

```

fd ∈ dom(h.fds) ∧
fid = h.fds[fd] ∧
h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
i'1 ∈ auto_outroute(i2, sock.is1, h.rttab, h.ifds) ∧
p'1 ∈ autobind(sock.ps1, (proto_of sock.pr), h.socks) ∧
(if sock.ps1 = * then bound' = sid :: bound else bound' = bound) ∧
(proto_of sock.pr = PROTO_UDP ⇒ ¬(linux_arch h.arch ∨ windows_arch h.arch)) ∧
(∃(sid', s) :: socks \ sid.
  s.is1 = ↑ i'1 ∧ s.ps1 = ↑ p'1 ∧
  s.is2 = ↑ i2 ∧ s.ps2 = ↑ p2 ∧
  proto_eq sock.pr s.pr) ∧
(if proto_of sock.pr = PROTO_UDP then
  if sock.is2 = * then is'1 = sock.is1 ∧ is'2 = * ∧ ps'2 = *
  else is'1 = * ∧ is'2 = * ∧ ps'2 = *
else is'1 = sock.is1 ∧ is'2 = sock.is2 ∧ ps'2 = sock.ps2)

```

Description

From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made where fd refers to a socket $sock$ identified by sid . The socket is either bound to local port $\uparrow p_1'$, or can be autobound to port $\uparrow p_1'$. The socket either has its local IP address set to $\uparrow i_1'$ or else its local IP address is unset but there exists an IP address i_1' for one of the host's interfaces which has a route to i_2 . There exists another socket s in the host's finite map of sockets, identified by sid' , that has as its binding quad $(\uparrow i_1', \uparrow p_1', \uparrow i_2, \uparrow p_2)$.

A $tid\cdot\text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EADDRINUSE})$: there is already another socket with the same local address connected to the peer address $(\uparrow i_2, \uparrow p_2)$. The socket's local port is set to $\uparrow p_1'$; if this was accomplished by autobinding then sid is appended to the head of $bound$, the host's list of bound sockets, to create a new list $bound'$. If $sock$ is a TCP socket then its is_1 , is_2 , and ps_2 fields are unchanged. If $sock$ is a UDP socket on FreeBSD then if its peer IP address was set, its local IP address will be unset: $is_1' = *$, otherwise its local IP address will stay as it was: $is_1' = sock.is_1$; its peer IP address and port will both be unset: $is_2' = * \wedge ps_2' = *$.

Variations

| | |
|-------|--|
| Linux | This rule does not apply to UDP sockets: Linux allows two UDP sockets to have the same binding quad. |
| WinXP | This rule does not apply to UDP sockets: WinXP allows two UDP sockets to have the same binding quad. |

connect_5c **all: fast fail** Fail with EADDRNOTAVAIL: no ephemeral ports left

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket$$

$$\frac{tid\cdot\text{connect}(fd, i_2, \uparrow p_2)}{\rightarrow} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EADDRNOTAVAIL}))_{\text{sched_timer}}) \rrbracket$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

$$fid = h.fds[fd] \wedge$$

$$h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$(h.socks[sid]).ps_1 = * \wedge$$

$$\text{autobind}(*, (\text{proto_of}(h.socks[sid]).pr), h.socks) = \emptyset$$

Description

From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ is made. fd refers to a socket identified by sid which is not bound to a local port. There are no ephemeral ports available to autobind to so the call fails with an EADDRNOTAVAIL error.

A $tid\cdot\text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EADDRNOTAVAIL})$.

connect_5d **tcp: block** Block, entering state Connect2: connection attempt already in progress and connect called with blocking semantics

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid\cdot\text{connect}(fd, i_2, \uparrow p_2)} h \llbracket ts := ts \oplus (tid \mapsto (\text{CONNECT2}(sid))_{\text{never_timer}}) \rrbracket$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

$$fid = h.fds[fd] \wedge$$

$$h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge$$

$$ff.b(\text{O_NONBLOCK}) = \mathbf{F} \wedge$$

$$\text{linux_arch } h.arch \wedge$$

$$tcp_sock.st \in \{\text{SYN_SENT}; \text{SYN_RECEIVED}\}$$

Description

From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made. fd refers to a TCP socket identified by sid which is in state SYN_SENT or SYN_RECEIVED: in other words, a connection attempt is already in progress for the socket (this could be an asynchronous connection attempt or one in another thread). The open file description referred to by fd does not have its O_NONBLOCK flag set so the call blocks, awaiting completion of the original connection attempt.

A $tid \cdot \text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state CONNECT2(sid).

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| WinXP | This rule does not apply. |

connect_6 **tcp: fast fail** Fail with EINVAL: socket has been shutdown for writing

$$\begin{aligned} & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ & socks := socks \oplus \\ & \quad [(sid, sock \langle cantsndmore := \mathbf{T}; pr := \text{TCP_PROTO}(tcp \langle st := \text{CLOSED} \rangle))] \rangle \\ \underline{tid \cdot \text{connect}(fd, i_2, \uparrow p_2)} \rightarrow & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}); \\ & socks := socks \oplus \\ & \quad [(sid, sock \langle cantsndmore := \mathbf{T}; pr := \text{TCP_PROTO}(tcp \langle st := \text{CLOSED} \rangle))] \rangle \end{aligned}$$

bsd_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$

Description

On FreeBSD, from thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, \uparrow p_2)$ call is made. fd refers to a TCP socket $sock$ identified by sid which is in state CLOSED and has been shutdown for writing.

A $tid \cdot \text{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

| | |
|-------|---------------------------|
| Posix | This rule does not apply. |
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

connect_7 **udp: fast succeed** Set peer address on socket with binding quad $*, ps_1, *, *$

$$\begin{aligned} & h_0 \\ \underline{tid \cdot \text{connect}(fd, i_2, ps_2)} \rightarrow \end{aligned}$$

$$\begin{aligned}
& h_0 \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
& \text{socks} := \text{socks} \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i'_1, \uparrow p'_1, \uparrow i_2, ps_2, es, \text{cantsndmore}', \text{cantrcvmore}, \text{UDP_PROTO}(udp)))]); \\
& \text{bound} := \text{bound} \\
& \rangle \\
\\
& h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
& \quad \text{socks} := \text{socks} \oplus \\
& \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, *, ps_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \\
& \quad \rangle \wedge \\
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
& p'_1 \in \text{autobind}(ps_1, \text{PROTO_UDP}, h_0.socks) \wedge \\
& (\mathbf{if} \ ps_1 = * \ \mathbf{then} \ \text{bound} = sid :: h_0.\text{bound} \ \mathbf{else} \ \text{bound} = h_0.\text{bound}) \wedge \\
& i'_1 \in \text{auto_outroute}(i_2, *, h_0.rttab, h_0.ifds) \wedge \\
& \neg(\exists(sid', s) :: (h_0.socks \setminus sid). \\
& \quad s.is_1 = \uparrow i'_1 \wedge s.ps_1 = \uparrow p'_1 \wedge \\
& \quad s.is_2 = \uparrow i_2 \wedge s.ps_2 = ps_2 \wedge \\
& \quad \text{proto_of } s.pr = \text{PROTO_UDP} \wedge \\
& \quad \text{bsd_arch } h.arch) \wedge \\
& (\text{bsd_arch } h.arch \implies ps_2 \neq * \wedge es = *) \wedge \\
& (\mathbf{if} \ \text{windows_arch } h.arch \ \mathbf{then} \ \text{cantsndmore}' = \mathbf{F} \\
& \ \mathbf{else} \ \text{cantsndmore}' = \text{cantsndmore})
\end{aligned}$$

Description

Consider a UDP socket sid , referenced by fd , with no local IP or peer address set. From thread tid , which is in the RUN state, a $\text{connect}(fd, i_2, ps_2)$ call is made. The socket's local port is either set to p'_1 , or it is unset and can be autobound to a local ephemeral port p'_1 . The local IP address can be set to i'_1 which is the primary IP address for an interface with a route to i_2 .

A $tid.\text{connect}(fd, i_2, ps_2)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. The socket's local address is set to $(\uparrow i'_1, \uparrow p'_1)$, and its peer address is set to $(\uparrow i_2, ps_2)$. If the socket's local port was autobound then sid is placed at the head of the host's list of bound sockets: $\text{bound} = sid :: h_0.\text{bound}$.

Variations

| | |
|---------|--|
| FreeBSD | As above, with the additional conditions that a foreign port is specified in the $\text{connect}()$ call: $ps_2 \neq *$, and there are no pending errors on the socket. Furthermore, there may be no other sockets in the host's finite map of sockets with the binding quad $(\uparrow i'_1, \uparrow p'_1, \uparrow i_2, ps_2)$. |
| WinXP | As above, except that the socket will not be shutdown for writing after the $\text{connect}()$ call has been made. |

connect_8 **udp: fast succeed** Set peer address on socket with local address set

$$\begin{aligned}
& h_0 \\
& \underline{tid.\text{connect}(fd, i, ps)} \\
& \quad h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
& \quad \text{socks} := \text{socks} \oplus \\
& \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i, ps, es, \text{cantsndmore}', \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \\
& \quad \rangle \\
& h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d);
\end{aligned}$$

$$\begin{aligned}
& socks := socks \oplus \\
& \quad [(sid, SOCK(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, UDP_PROTO(udp)))] \wedge \\
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h.files[fd] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge \\
& (\mathbf{bsd_arch} \ h.arch \implies ps \neq * \wedge es = *) \wedge \\
& (\mathbf{if} \ \mathbf{windows_arch} \ h.arch \ \mathbf{then} \ cantsndmore' = \mathbf{F} \\
& \ \mathbf{else} \ cantsndmore' = cantsndmore) \wedge \\
& \neg(\exists(sid', s) :: (h_0.socks \setminus sid). \\
& \quad s.is_1 = \uparrow i_1 \wedge s.ps_1 = \uparrow p_1 \wedge \\
& \quad s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge \\
& \quad \mathbf{proto_of} \ s.pr = \mathbf{PROTO_UDP} \wedge \\
& \quad \mathbf{bsd_arch} \ h.arch)
\end{aligned}$$

Description

Consider a UDP socket sid , referenced by fd , with local address set to $(\uparrow i_1, \uparrow p_1)$. Its peer address may or may not be set. From thread tid , which is in the RUN state, a $\mathbf{connect}(fd, i, ps)$ call is made.

The call succeeds: a $tid.\mathbf{connect}(fd, i, ps)$ transition is made, leaving the thread in state $\mathbf{RET}(\mathbf{OK}())$. The socket has its peer address set to $(\uparrow i, ps)$.

Variations

| | |
|---------|---|
| FreeBSD | As above, with the additional conditions that a foreign port is specified in the $\mathbf{connect}()$ call, $ps \neq *$, and there are no pending errors on the socket. Furthermore, there may be no other sockets in the host's finite map of sockets with the binding quad $(\uparrow i'_1, \uparrow p1', \uparrow i, ps)$. |
| WinXP | As above, with the additional effect that if the socket was shutdown for writing when the $\mathbf{connect}()$ call was made, it will no longer be shutdown for writing. |

connect_9 **udp: fast fail** Fail with EADDRNOTAVAIL: port must be specified in connect() call on FreeBSD

$$\begin{aligned}
& h \langle ts := ts \oplus (tid \mapsto (\mathbf{RUN})_d); \\
& \quad socks := socks \oplus \\
& \quad \quad [(sid, sock \langle pr := \mathbf{UDP_PROTO}(udp) \rangle)] \rangle \\
& \xrightarrow{tid.\mathbf{connect}(fd, i, *)} h \langle ts := ts \oplus (tid \mapsto (\mathbf{RET}(\mathbf{FAIL} \ \mathbf{EADDRNOTAVAIL}))_{\mathbf{sched_timer}}); \\
& \quad socks := socks \oplus \\
& \quad \quad [(sid, sock \langle is_1 := is_1; is_2 := *; ps_2 := *; pr := \mathbf{UDP_PROTO}(udp) \rangle)] \rangle
\end{aligned}$$

$$\begin{aligned}
& \mathbf{bsd_arch} \ h.arch \wedge \\
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h.files[fd] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge \\
& (\mathbf{if} \ sock.is_2 \neq * \ \mathbf{then} \ is_1 = * \ \mathbf{else} \ is_1 = sock.is_1)
\end{aligned}$$

Description

On FreeBSD, consider a UDP socket sid referenced by fd . From thread tid , which is in the RUN state, a $\mathbf{connect}(fd, i, *)$ call is made. Because no port is specified, the call fails with an EADDRNOTAVAIL error.

A $tid.\mathbf{connect}(fd, i, *)$ transition is made, leaving the thread state $\mathbf{RET}(\mathbf{FAIL} \ \mathbf{EADDRNOTAVAIL})$. The socket's peer address is cleared: $is_2 := *$ and $ps_2 := *$. Additionally, if the socket had its peer IP address set, $sock.is_2 \neq *$, then its local IP address will be cleared: $is_1 = *$; otherwise it remains the same: $is_1 = sock.is_1$.

Variations

| | |
|-------|---------------------------|
| Posix | This rule does not apply. |
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

connect_10 **udp: fast fail** Fail with pending error on FreeBSD, but still set peer address

$$h_0 \xrightarrow{tid \cdot \text{connect}(fd, i, ps)} h_0 \langle [ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}});$$

$$socks := socks \oplus$$

$$[(sid, sock \langle [is_2 := \uparrow i; ps_2 := ps; es := *; pr := \text{UDP_PROTO}(udp) \rangle]) \rangle]$$

bsd_arch $h.arch \wedge$
 $h_0 = h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d);$
 $socks := socks \oplus$
 $[(sid, sock \langle [es := \uparrow err; pr := \text{UDP_PROTO}(udp) \rangle]) \rangle] \rangle \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $ps \neq * \wedge$
 $\neg(\exists(sid', s) :: (h_0.socks \setminus sid).$
 $s.is_1 = sock.is_1 \wedge s.ps_1 = sock.ps_1 \wedge$
 $s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge$
 $\text{proto_of } s.pr = \text{PROTO_UDP})$

Description

On FreeBSD, consider a UDP socket sid , referenced by fd , with pending error err . From thread tid , which is in the RUN state, a $\text{connect}(fd, i, ps)$ call is made with $ps \neq *$. There is no other UDP socket on the host which has the same local address $sock.is_1, sock.ps_1$ as sid , and its peer address set to $\uparrow i, ps$. The call fails, returning the pending error err .

A $tid \cdot \text{connect}(fd, i, ps)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } err)$. The socket's peer address is set to $(\uparrow i, ps)$, and the error is cleared from the socket.

Variations

| | |
|-------|---------------------------|
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

15.5 disconnect() (TCP and UDP)

`disconnect` : $fd \rightarrow \text{unit}$

A call to $\text{disconnect}(fd)$, where fd is a file descriptor referring to a socket, removes the peer address for a UDP socket. If a UDP socket has peer address set to $(\uparrow i_2, \uparrow p_2)$ then it can only receive datagrams with source address (i_2, p_2) . Calling $\text{disconnect}()$ on the socket resets its peer address to $(*, *)$, and so it will be able to receive datagrams with any source address.

It does not make sense to disconnect a TCP socket in this way. Most supported architectures simply disallow disconnect on such a socket; however, Linux implements it as an abortive close (see *close_3* (p139)).

15.5.1 Errors

A call to `disconnect()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|---------------|---|
| EADDRNOTAVAIL | There are no ephemeral ports left for autobinding to. |
| EAFNOSUPPORT | The address family <code>AF_UNSPEC</code> is not supported. This can be the result for a successful <code>disconnect()</code> for a UDP socket. |
| EAGAIN | There are no ephemeral ports left for autobinding to. |
| EALREADY | A connection is already in progress. |
| EBADF | The file descriptor <code>fd</code> is an invalid file descriptor. |
| EISCONN | The socket is already connected. |
| ENOBUFS | No buffer space is available. |
| EOPNOTSUPP | The socket is listening and cannot be connected. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.5.2 Common cases

disconnect_1; return_1

15.5.3 API

`disconnect()` is a Posix `connect()` call with the address family set to `AF_UNSPEC`.

```
Posix:      int connect(int socket, const struct sockaddr *address,
                  socklen_t address_len);
FreeBSD:    int connect(int s, const struct sockaddr *name,
                  socklen_t namelen);
Linux:      int connect(int sockfd, const struct sockaddr *serv_addr,
                  socklen_t addrlen);
WinXP:      int connect(SOCKET s, const struct sockaddr* name,
                  int namelen);
```

In the Posix interface:

- `socket` is a file descriptor referring to a socket. This corresponds to the `fd` argument of the model `disconnect()`.
- `address` is a pointer to a location of size `address_len` containing a `sockaddr` structure which specifies the address to connect to. For a `disconnect()` call, the `sin_family` field of the `sockaddr` family must be set to `AF_UNSPEC`; other fields can be set to anything.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The Linux man-page states: "Unconnecting a socket by calling `connect` with a `AF_UNSPEC` address is not yet implemented." As a result, a `disconnect()` call always returns successfully on Linux.

The WinXP documentation states: "The default destination can be changed by simply calling `connect` again, even if the socket is already connected. Any datagrams queued for receipt are discarded if `name` is different from the previous `connect`." This implies that calling `disconnect()` will result in all datagrams on the socket's receive queue; however, this is not the case: no datagrams are discarded.

15.5.4 Summary

| | | |
|---------------------|--------------------------|---|
| <i>disconnect_4</i> | tcp: fast fail | Fail with EAFNOSUPPORT: address family not supported; EOPNOTSUPP: operation not supported; EALREADY: connection already in progress; or EISCONN: socket already connected |
| <i>disconnect_5</i> | tcp: fast fail | Succeed on Linux, possibly dropping the connection |
| <i>disconnect_1</i> | udp: fast succeed | Unset socket's peer address |
| <i>disconnect_2</i> | udp: fast succeed | Unset socket's peer address and autobind local port |
| <i>disconnect_3</i> | udp: fast fail | Fail with EAGAIN, EADDRNOTAVAIL, or ENOBUFS: there are no ephemeral ports left |

15.5.5 Rules

disconnect_4 **tcp: fast fail** Fail with EAFNOSUPPORT: address family not supported; EOPNOTSUPP: operation not supported; EALREADY: connection already in progress; or EISCONN: socket already connected

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{disconnect}(fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched.timer}}) \rangle$$

$fd \in \text{dom}(h.fds) \wedge$

$fid = h.fds[fd] \wedge$

$h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$

$\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge$

$\neg(\text{linux_arch } h.arch) \wedge$

case *tcp_sock.st* **of**

CLOSED \rightarrow **if** *bsd_arch h.arch* **then**

if *tcp_sock.cb.bsd_cantconnect* = **T** **then** *err* = EINVAL

else *err* = EAFNOSUPPORT

else *err* = EAFNOSUPPORT **||**

LISTEN \rightarrow **if** *windows_arch h.arch* **then** *err* = EAFNOSUPPORT (* socket is listening *)

else if *bsd_arch h.arch* **then** *err* = EOPNOTSUPP

else ASSERTION_FAILURE“disconnect_4:1” **||** (* never happen *)

SYN_SENT \rightarrow *err* = EALREADY **||** (* connection already in progress *)

SYN_RECEIVED \rightarrow *err* = EALREADY **||** (* connection already in progress *)

ESTABLISHED \rightarrow *err* = EISCONN **||** (* socket already connected *)

TIME_WAIT \rightarrow **if** *windows_arch h.arch* **then** *err* = EISCONN

else if *bsd_arch h.arch* **then** *err* = EAFNOSUPPORT

else ASSERTION_FAILURE“disconnect_4:2” **||** (* never happen *)

_1 \rightarrow *err* = EISCONN (* all other states *)

Description

Consider a TCP socket *sid* referenced by *fd* on a non-Linux architecture. From thread *tid*, which is in the RUN state, a `disconnect(fd)` call is made. The call fails with an error *err* which depends on the state of the socket: If the socket is in the CLOSED state then it fails with EAFNOSUPPORT, except if on FreeBSD its *bsd_cantconnect* flag is set, in which case it fails with EINVAL; if it is in the LISTEN state the error is EAFNOSUPPORT on WinXP and EOPNOTSUPP on FreeBSD; if it is in the SYN_SENT or SYN_RECEIVED state the error is EALREADY; if it is in the ESTABLISHED state the error is EISCONN; if it is in the TIME_WAIT state the error is EISCONN on WinXP and EAFNOSUPPORT on FreeBSD; in all other states the error is EISCONN.

A `tid.disconnect(fd)` transition is made, leaving the thread state `RET(FAIL err)` where *err* is one of the above errors.

Variations

| | |
|-------|---------------------------|
| Linux | This rule does not apply. |
|-------|---------------------------|

disconnect_5 **tcp: fast fail** Succeed on Linux, possibly dropping the connection

$$h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$socks := socks \oplus [(sid, sock)];$$

$$oq := oq] \rangle \xrightarrow{tid \cdot \text{disconnect}(fd)} h \langle [ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}});$$

$$socks := socks \oplus [(sid, sock')];$$

$$oq := oq'] \rangle$$

linux_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $\text{TCP_PROTO}(tcp_sock) = sock.pr \wedge$
(if $tcp_sock.st \in \{\text{SYN_RECEIVED}; \text{ESTABLISHED}; \text{FIN_WAIT_1}; \text{FIN_WAIT_2}; \text{CLOSE_WAIT}\}$ **then**
 $tcp_drop_and_close \ h.arch * sock(sock', outsegs) \wedge$
 $enqueue_and_ignore_fail \ h.arch \ h.rttab \ h.ifds \ outsegs \ oq \ oq'$
else
 $sock = sock' \wedge$
 $oq = oq'$)

Description

On Linux, consider a TCP socket sid , referenced by fd . From thread tid , which is in the RUN state, a $\text{disconnect}(fd)$ call is made and succeeds.

A $tid \cdot \text{disconnect}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. If the socket is in the SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, or CLOSE_WAIT state then the connection is dropped, a RST segment is constructed, $outsegs$, which may be placed on the host's outqueue, oq , resulting in new outqueue oq' . If the socket is in any other state then it remains unchanged, as does the host's outqueue.

Model details

Note that $\text{disconnect}()$ has not been properly implemented on Linux yet so it will always succeed.

Variations

| | |
|---------|---------------------------|
| Posix | This rule does not apply. |
| FreeBSD | This rule does not apply. |
| WinXP | This rule does not apply. |

disconnect_1 **udp: fast succeed** Unset socket's peer address

$$h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$socks := socks \oplus$$

$$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))]$$

$$\rangle$$

$$\xrightarrow{tid \cdot \text{disconnect}(fd)}$$

$$\begin{aligned}
& h \langle ts := ts_ \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched_timer}}); \\
& \text{socks} := \text{socks} \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, *, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \\
& \rangle
\end{aligned}$$

$$\begin{aligned}
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
& (\text{if linux_arch } h.arch \text{ then } ret = \text{OK}() \\
& \text{else if windows_arch } h.arch \wedge \exists i'_2.is_2 = \uparrow i'_2 \text{ then } ret = \text{OK}() \\
& \text{else } ret = \text{FAIL EAFNOSUPPORT})
\end{aligned}$$

Description

Consider a UDP socket sid referenced by fd with $(is_1, \uparrow p_1, is_2, ps_2)$ as its binding quad. From thread tid , which is in the RUN state, a $\text{disconnect}(fd)$ call is made. On Linux the call succeeds; on WinXP if the socket had its peer IP address set then the call succeeds, otherwise it fails with an EAFNOSUPPORT error; on FreeBSD the call fails with an EAFNOSUPPORT error.

A $tid \cdot \text{disconnect}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$ or $\text{RET}(\text{FAIL EAFNOSUPPORT})$. The socket has its peer address set to $(*, *)$, and its local IP address set to $*$. The local port, p_1 , is left in place.

Variations

| | |
|---------|---|
| FreeBSD | As above: the call fails with an EAFNOSUPPORT error. |
| Linux | As above: the call succeeds. |
| WinXP | As above: the call succeeds if the socket had a peer IP address set, or fails with an EAFNOSUPPORT error otherwise. |

disconnect_2 **udp: fast succeed** Unset socket's peer address and autobind local port

h_0

$tid \cdot \text{disconnect } fd$

$$\begin{aligned}
& h_0 \langle ts := ts_ \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched_timer}}); \\
& \text{socks} := \text{socks} \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, *, \uparrow p_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \\
& \text{bound} := sid :: h_0.\text{bound} \rangle
\end{aligned}$$

$$\begin{aligned}
& h_0 = h \langle ts := ts_ \oplus (tid \mapsto (\text{RUN})_d); \\
& \quad \text{socks} := \text{socks} \oplus \\
& \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \rangle \wedge \\
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
& p_1 \in \text{autobind}(*, \text{PROTO_UDP}, h_0.\text{socks}) \wedge \\
& (\text{if linux_arch } h.arch \text{ then } ret = \text{OK}() \\
& \text{else } ret = (\text{FAIL EAFNOSUPPORT}))
\end{aligned}$$

Description

Consider a UDP socket sid referenced by fd and with binding quad $(*, *, *, *)$. From thread tid , which is in the RUN state, a $\text{disconnect}(fd)$ call is made. The call succeeds on Linux and fails with an EAFNOSUPPORT error on FreeBSD and WinXP.

A $tid\text{-disconnect}(fd)$ transition is made, leaving the thread either in state $\text{RET}(\text{OK}())$, or in state $\text{RET}(\text{FAIL EAFNOSUPPORT})$. The socket is autobound to a local ephemeral port $p1'$, and sid is placed on the head of the host's list of bound sockets.

Variations

| | |
|---------|--|
| FreeBSD | As above: the call fails with an EAFNOSUPPORT error. |
| Linux | As above: the call succeeds. |
| WinXP | As above: the call fails with an EAFNOSUPPORT error. |

disconnect_3 udp: fast fail Fail with EAGAIN, EADDRNOTAVAIL, or ENOBUFS: there are no ephemeral ports left

$$h_0 \xrightarrow{tid\text{-disconnect } fd} h_0 \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rangle \rangle$$

$$h_0 = h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{socks} := \text{socks} \oplus [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp))] \rangle \rangle \wedge$$

$$fd \in \text{dom}(h.fds) \wedge$$

$$fid = h.fds[fd] \wedge$$

$$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$\text{autobind}(*, \text{PROTO_UDP}, h_0.\text{socks}) = \emptyset \wedge$$

$$e \in \{\text{EAGAIN}; \text{EADDRNOTAVAIL}; \text{ENOBUFS}\}$$

Description

Consider a UDP socket sid referenced by fd and with binding quad $*, *, *, *$. From thread tid , which is in the RUN state, a $\text{disconnect}(fd)$ call is made. There are no ephemeral ports left, so the socket cannot be autobound to a local port. The call fails with an error: EAGAIN, EADDRNOTAVAIL, or ENOBUFS.

A $tid\text{-disconnect}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } e)$ where e is one of the above errors.

15.6 dup() (TCP and UDP)

$\text{dup} : fd \rightarrow fd$

A call to $\text{dup}(fd)$ creates and returns a new file descriptor referring to the open file description referred to by the file descriptor fd . A successful $\text{dup}()$ call will return the least numbered free file descriptor. The call will only fail if there are no more free file descriptors, or fd is not a valid file descriptor.

15.6.1 Errors

A call to $\text{dup}()$ can fail with the errors below, in which case the corresponding exception is raised:

| | |
|--------|--|
| EMFILE | There are no more file descriptors available. |
| EBADF | The file descriptor passed is not a valid file descriptor. |

15.6.2 Common cases

dup_1; *return_1*

15.6.3 API

Posix: `int dup(int fildes);`
 FreeBSD: `int dup(int oldd);`
 Linux: `int dup(int oldfd);`
 In the Posix interface:

- `fildes` is a file descriptor referring to the open file description for which another file descriptor is to be created for. This corresponds to the `fd` argument of the model `dup()`.
- The returned `int` is either non-negative to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. If the call is successful then the returned `int` is the new file descriptor corresponding to the `fd` return type of the model `dup()`.

The FreeBSD and Linux interfaces are similar. This call does not exist on WinXP.

15.6.4 Summary

| | | |
|--------------|--------------------------|--|
| <i>dup_1</i> | all: fast succeed | Successfully duplicate file descriptor |
| <i>dup_2</i> | all: fast fail | Fail with EMFILE: no more file descriptors available |

15.6.5 Rules

| |
|--|
| <div style="border-bottom: 1px solid black; margin-bottom: 10px;"></div> <p><i>dup_1</i> all: fast succeed Successfully duplicate file descriptor</p> $h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \rangle \rangle_{fds := fds} \xrightarrow{tid \cdot \text{dup}(fd)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } fd')_{\text{sched_timer}}); \rangle \rangle_{fds := fds'}$ <p> $unix_arch \ h.arch \wedge$ $fd \in \text{dom}(h.fds) \wedge$ $fid = h.fds[fd] \wedge$ $nextfd \ h.arch \ fds \ fd' \wedge$ $fd' < \text{OPEN_MAX_FD} \wedge$ $fds' = fds \oplus (fd', fid)$ </p> |
|--|

Description

From thread *tid*, which is in the RUN state, a `dup(fd)` call is made where *fd* is a file descriptor referring to an open file description identified by *fid*. A new file descriptor, *fd'* can be created in an architecture-specific way according to the `nextfd (p??)` function. *fd'* is less than the maximum open file descriptor, `OPEN_MAX_FD`. The call succeeds returning *fd'*.

A `tid·dup(fd)` transition is made, leaving the thread state `RET(OK fd')`. The host's finite map of file descriptors, *fds*, is extended to map the new file descriptor *fd'* to the file identifier *fid*, which results in a new finite map of file descriptors *fds'* for the host.

Variations

| | |
|-------|---|
| WinXP | This rule does not apply: there is no <code>dup()</code> call on WinXP. |
|-------|---|

dup_2 **all: fast fail** Fail with EMFILE: no more file descriptors available

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid \cdot \text{dup}(fd)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EMFILE}))_{\text{sched_timer}}) \rrbracket$$

unix_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $(\text{card}(\text{dom}(h.fds)) + 1) \geq \text{OPEN_MAX}$

Description

From thread tid , which is in the RUN state, a $\text{dup}(fd)$ call is made where fd is a valid file descriptor: it has an entry in the host's finite map of file descriptors, $h.fds$. Creating another file descriptor would cause the number of open file descriptors to be greater than or equal to the maximum number of open file descriptors, OPEN_MAX. The call fails with an EMFILE error.

A $tid \cdot \text{dup}(fd)$ transition is made, leaving the thread state RET(FAIL EMFILE).

Variations

| | |
|-------|--|
| WinXP | This rule does not apply: there is no dup() call on WinXP. |
|-------|--|

15.7 dupfd() (TCP and UDP)

$\text{dupfd} : \text{fd} * \text{int} \rightarrow \text{fd}$

A call to $\text{dupfd}(\text{fd}, n)$ creates and returns a new file descriptor referring to the open file description referred to by the file descriptor fd .

A successful $\text{dupfd}()$ call will return the least free file descriptor greater than or equal to n . The call will fail if n is negative or greater than the maximum allowed file descriptor, OPEN_MAX; if the file descriptor fd is not a valid file descriptor; or if there are no more file descriptors available.

15.7.1 Errors

A call to $\text{dupfd}()$ can fail with the errors below, in which case the corresponding exception is raised:

| | |
|--------|---|
| EINVAL | The requested file descriptor is invalid: it is negative or greater than the maximum allowed. |
| EMFILE | There are no more file descriptors available. |
| EBADF | The file descriptor passed is not a valid file descriptor. |

15.7.2 Common cases

dupfd_1; return_1

15.7.3 API

$\text{dupfd}()$ is Posix $\text{fcntl}()$ using the F_DUPFD command:

```
Posix:    int fcntl(int fildes, int cmd, int arg);
FreeBSD:  int fcntl(int fd, int cmd, int arg);
Linux:    int fcntl(int fd, int cmd, long arg);
```


dupfd_3 **all: fast fail** Fail with EINVAL: n is negative or greater than the maximum allowed file descriptor

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid \cdot \text{dupfd}(fd, n)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}) \rangle \rangle$$

unix_arch $h.arch \wedge$
 $n < 0 \vee \mathbf{num} \ n \geq \text{OPEN_MAX} \wedge$
 $err = (\mathbf{if} \ \text{bsd_arch} \ h.arch \ \mathbf{then} \ \text{EBADF} \ \mathbf{else} \ \text{EINVAL})$

Description

From thread tid , which is in the RUN state, a $\text{dupfd}(fd, n)$ call is made. n is either negative or greater than the maximum number of open file descriptors, OPEN_MAX. The call fails with an EINVAL error.

A $tid \cdot \text{dupfd}(fd, n)$ transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

| | |
|---------|--|
| WinXP | This call does not apply: there is no dupfd() call on WinXP. |
| FreeBSD | On BSD the error EBADF is returned. |

dupfd_4 **all: fast fail** Fail with EMFILE: no more file descriptors available

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid \cdot \text{dupfd}(fd, n)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } \text{EMFILE}))_{\text{sched_timer}}) \rangle \rangle$$

unix_arch $h.arch \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $n \geq 0 \wedge$
 $fd' = \text{FD}(\mathbf{least} \ n'. \ \mathbf{num} \ n \leq n' \wedge \text{OPEN_MAX_FD} \leq \text{FD } n' \wedge \text{FD } n' \notin \mathbf{dom}(h.fds))$

Description

From thread tid , which is in the RUN state, a $\text{dupfd}(fd, n)$ call is made. fd is a file descriptor referring to open file description fid and n is non-negative. The least file descriptor fd' that is greater than or equal to n is greater than or equal to the maximum open file descriptor, OPEN_MAX_FD. The call fails with an EMFILE error.

A $tid \cdot \text{dupfd}(fd, n)$ transition is made, leaving the thread state RET(FAIL EMFILE).

Variations

| | |
|-------|--|
| WinXP | This rule does not apply: there is no dupfd() call on WinXP. |
|-------|--|

15.8 getfileflags() (TCP and UDP)

getfileflags : fd \rightarrow filebflag list

A call to getfileflags(fd) returns a list of the file flags currently set for the file which fd refers to. The possible file flags are:

- O_ASYNC Reports whether signal driven I/O is enabled.

- `O_NONBLOCK` Reports whether a socket is non-blocking.

15.8.1 Errors

A call to `getfileflags()` can fail with the error below, in which case the corresponding exception is raised:

| | |
|--------------------|--|
| <code>EBADF</code> | The file descriptor passed is not a valid file descriptor. |
|--------------------|--|

15.8.2 Common cases

A call to `getfileflags()` is made, returning the flags set: *getfileflags_1*; *return_1*

15.8.3 API

`getfileflags()` is Posix `fcntl(fd,F_GETFL)`. On WinXP it is `ioctlsocket()` with the `FIONBIO` command.

```
Posix:      int fcntl(int fildes, int cmd, ...);
FreeBSD:   int fcntl(int fd, int cmd, ...);
Linux:     int fcntl(int fd, int cmd);
WinXP:     int ioctlsocket(SOCKET s, long cmd, u_long* argp)
```

In the Posix interface:

- `fildes` is a file descriptor for the file to retrieve flags from. It corresponds to the `fd` argument of the model `getfileflags()`. On WinXP the `s` is a socket descriptor corresponding to the `fd` argument of the model `getfileflags()`.
- `cmd` is a command to perform an operation on the file. This is set to `F_GETFL` for the model `getfileflags()`. On WinXP, `cmd` is set to `FIONBIO` to get the `O_NONBLOCK` flag; there is no `O_ASYNC` flag on WinXP.
- The call takes a variable number of arguments. For the model `getfileflags()` only the two arguments described above are needed.
- If the call succeeds the returned `int` represents the file flags that are set corresponding to the `filebflag` list return type of the model `getfileflags()`. If the returned `int` is `-1` then an error has occurred in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR` with the actual error code available through a call to `WSAGetLastError()`.

15.8.4 Model details

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `WSAENOTSOCK` is a possible error on WinXP as the `ioctlsocket()` call is specific to a socket. In the model the `getfileflags()` call is performed on a file.

15.8.5 Summary

| | | |
|-----------------------|--------------------------|--|
| <i>getfileflags_1</i> | all: fast succeed | Return list of file flags currently set for an open file description |
|-----------------------|--------------------------|--|

15.8.6 Rules

getfileflags_1 **all: fast succeed** Return list of file flags currently set for an open file description

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{getfileflags}(fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } flags))_{\text{sched_timer}}) \rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(ft, ff) \wedge$
 $flags \in \text{ORDERINGS } ff.b$

Description

From thread *tid*, which is in the RUN state, a `getfileflags(fd)` call is made. *fd* refers to a file description `FILE(ft, ff)` where *ff* is the file flags that are set. The call succeeds, returning *flags* which is a list representing some ordering of the boolean file flags *ff.b* in *ff*.

A `tid.getfileflags(fd)` transition is made, leaving the thread state `RET(OK(flags))`.

15.9 getifaddrs() (TCP and UDP)

`getifaddrs` : `unit` \rightarrow (*ifid* * ip * ip list * *netmask*)list

A call to `getifaddrs()` returns the interface information for a host. For each interface a tuple is constructed consisting of: the interface name, the primary IP address for the interface, the auxiliary IP addresses for the interface, and the subnet mask for the interface. A list is constructed with one tuple for each interface, and this is the return value of the call to `getifaddrs()`.

15.9.1 Errors

| | |
|-------|--|
| EINTR | The system was interrupted by a caught signal. |
| EBADF | The file descriptor passed is not a valid file descriptor. |

15.9.2 Common cases

getifaddrs_1; *return_1*

15.9.3 API

`getifaddrs()` is two calls to Posix `ioctl()`: one with the `SIOCGIFCONF` request and one with the `SIOCGIFNETMASK` request. On FreeBSD there is a specific `getifaddrs()` call. On WinXP the `getifaddrs()` call does not exist.

Posix: `int ioctl(int fildes, int request, ... /* arg */);`

FreeBSD: `int getifaddrs(struct ifaddrs **ifap);`

Linux: `int ioctl(int d, int request, ...);`

In the Posix interface:

- `fildes` is a file descriptor. There is no corresponding argument in the model `getifaddrs()`.
- `request` is the operation to perform on the file. When `request` is `SIOCGIFCONF` the list of all interfaces is returned; when it is `SIOCNETMASK` the subnet mask is returned for an interface.
- The function takes a variable number of arguments. When `request` is `SIOCGIFCONF` there is a third argument: a pointer to a location to store a linked-list of the interfaces; when it is `SIOCGIFNETMASK` it is a pointer to a structure containing the interface and it is filled in with the subnet mask for that interface.
- The returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`.

To construct the return value of type $(ifid * ip * ip\ list * netmask)list$, the interface name and the IP addresses associated with it are obtained from the call to `ioctl()` using `SIOCGIFCONF`, and then the subnet mask for each interface is obtained from a call to `ioctl()` using `SIOCGIFNETMASK`.

On FreeBSD the `ifap` argument to `getifaddrs()` is a pointer to a location to store a linked list of the interface information in, corresponding to the return type of the model `getifaddrs()`.

15.9.4 Model details

Any of the errors possible when making an `ioctl()` call are possible: `EIO`, `ENOTTY`, `ENXIO`, and `ENODEV`. None of these are modelled.

Note that the Posix interface admits the possibility that the interfaces will change between the two calls, whereas in the model interface the `getifaddrs()` call is atomic.

15.9.5 Summary

`getifaddrs_1` **all: fast succeed** Successfully return host interface information

15.9.6 Rules

`getifaddrs_1` **all: fast succeed** Successfully return host interface information

$$h \ ts := ts \oplus (tid \mapsto (RUN)_d) \xrightarrow{tid \cdot \text{getifaddrs}()} h \ ts := ts \oplus (tid \mapsto (RET(OK\ iflist))_{\text{sched_timer}})$$

$ifidlist \in \text{ORDERINGS } ifidset \wedge$
 $\text{length } ifidlist = \text{length } iflist \wedge$

$ifidset = \{(ifid, hifd) \mid$
 $\quad ifid \in \text{dom}(h.ifds) \wedge$
 $\quad hifd = h.ifds[ifid]\} \wedge$

every $\mathbf{I}(\text{map2}(\lambda(ifid, hifd)(ifid', primary, ipslist, netmask).(ifid' = ifid \wedge$
 $\quad primary = hifd.primary \wedge$
 $\quad ipslist \in \text{ORDERINGS } hifd.ipset \wedge$
 $\quad netmask = hifd.netmask))$
 $\quad ifidlist\ iflist)$

Description

On a Unix architecture, from thread `tid`, which is in the `RUN` state, a `getifaddrs()` call is made. The call succeeds, returning `iflist` which is a list of tuples: one for each interface on the host. Each tuple consists of: the interface name; the primary IP address for the interface; a list of the other IP addresses for the interface; and the netmask for the interface.

A `tid.getifaddrs()` transition is made, leaving the thread state `RET(OK iflist)`.

Variations

| | |
|-------|------------------------------------|
| WinXP | This call does not exist on WinXP. |
|-------|------------------------------------|

15.10 getpeername() (TCP and UDP)

`getpeername` : $fd \rightarrow (ip * port)$

A call to `getpeername(fd)` returns the peer address of the socket referred to by file descriptor `fd`. If the file descriptor refers to a socket `sock` then a successful call will return (i_2, p_2) where $sock.is_2 = \uparrow i_2$, and $sock.ps_2 = \uparrow p_2$.

15.10.1 Errors

A call to `getpeername()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------|--|
| ENOTCONN | Socket not connected to a peer. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.10.2 Common cases

getpeername_1; return_1

15.10.3 API

```
Posix:    int getpeername(int socket, struct sockaddr *restrict address,
                    socklen_t *restrict address_len);
FreeBSD:  int getpeername(int s, struct sockaddr *name,
                    socklen_t *namelen);
Linux:    int getpeername(int s, struct sockaddr *name,
                    socklen_t *namelen);
WinXP:    int getpeername(SOCKET s, struct sockaddr* name,
                    int* namelen);
```

In the Posix interface:

- `socket` is a file descriptor referring to the socket to get the peer address of, corresponding to the `fd` argument in the model `getpeername()`.
- `address` is a pointer to a `sockaddr` structure of length `address_len`, which contains the peer address of the socket upon return. These two correspond to the `(ip*port)` return type of the model `getpeername()`. The `sin_addr.s_addr` field of the `address` structure holds the peer IP address, corresponding to the `ip` in the return tuple; the `sin_port` field of the `address` structure holds the peer port, corresponding to the `port` in the return tuple.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.10.4 Model details

The following errors are not modelled:

- According to the FreeBSD man page for `getpeername()`, `ECONNRESET` can be returned if the connection has been reset by the peer. This behaviour has not been observed in any tests.
- On FreeBSD, Linux, and WinXP, `EFAULT` can be returned if the `name` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `getpeername()` that is excluded by the clean interface used in the model `getpeername()`.
- In Posix, `EINVAL` can be returned if the socket has been shutdown; none of the implementations in the model return this error from a `getpeername()` call.
- In Posix, `EOPNOTSUPP` is returned if the `getpeername()` operation is not supported by the protocol. Both TCP and UDP support this operation.

- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.10.5 Summary

getpeername_1 **all: fast succeed** Successfully return socket's peer address
getpeername_2 **all: fast fail** Fail with ENOTCONN: socket not connected to a peer

15.10.6 Rules

getpeername_1 **all: fast succeed** **Successfully return socket's peer address**

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid.\text{getpeername}(fd)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(i_2, p_2)))_{\text{sched_timer}}) \rangle \rangle$$

$$\begin{aligned} & fd \in \mathbf{dom}(h.fds) \wedge \\ & fid = h.fds[fd] \wedge \\ & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ & sock = h.socks[sid] \wedge \\ & sock.is_2 = \uparrow i_2 \wedge \\ & (sock.ps_2 = \uparrow p_2 \vee (\text{windows_arch } h.arch \wedge sock.ps_2 = * \wedge \\ & \quad (p_2 = \text{PORT } 0) \wedge \text{proto_of } sock.pr = \text{PROTO_UDP})) \wedge \\ & ((\forall tcp_sock. sock.pr = \text{TCP_PROTO}(tcp_sock) \implies \\ & \quad tcp_sock.st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}; \text{LAST_ACK}; \\ & \quad \quad \text{FIN_WAIT_1}; \text{CLOSING}\} \vee \\ & \quad (\neg sock.cantrcvmore \wedge tcp_sock.st = \text{FIN_WAIT_2}) \vee \\ & \quad (\text{linux_arch } h.arch \wedge tcp_sock.st = \text{SYN_RECEIVED}) \vee \\ & \quad (* \text{ BSD listen bug } *) \\ & \quad (\text{bsd_arch } h.arch \wedge tcp_sock.st = \text{LISTEN})) \vee \\ & \text{windows_arch } h.arch) \end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a *getpeername(fd)* call is made. *fd* refers to a socket *sock*, identified by *sid*, which has its peer IP address set to $\uparrow i_2$ and its peer port address set to $\uparrow p_2$. If *sock* is a TCP socket then either it is in state ESTABLISHED, CLOSE_WAIT, LAST_ACK, FIN_WAIT_1, or CLOSING; or it is in state FIN_WAIT_2 and is not shutdown for reading. The call succeeds, returning (i_2, p_2) , the socket's peer address.

A *tid.getpeername(fd)* transition is made, leaving the thread state $\text{RET}(\text{OK}(i_2, p_2))$.

Variations

| | |
|---------|--|
| FreeBSD | If <i>sock</i> is a TCP socket then it may be in state LISTEN; this is due to the FreeBSD bug that allows <code>listen()</code> to be called on a synchronised socket. |
| Linux | If <i>sock</i> is a TCP socket then it may also be in state SYN_RECEIVED. |
| WinXP | If <i>sock</i> is a UDP socket and has no peer port set, $sock.ps_2 = *$ then the call may still succeed with $p_2 = \text{PORT } 0$. Additionally, if <i>sock</i> is a TCP socket then it may be in any state. |

getpeername_2 **all: fast fail** Fail with ENOTCONN: socket not connected to a peer

$$\frac{h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket}{tid \cdot \text{getpeername}(fd) \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOTCONN}))_{\text{sched_timer}}) \rrbracket}$$

$$\begin{aligned} & fd \in \mathbf{dom}(h.fds) \wedge \\ & fid = h.fds[fd] \wedge \\ & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ & sock = h.socks[sid] \wedge \\ & \neg(sock.is_2 \neq * \wedge \\ & \quad (sock.ps_2 \neq * \vee (\text{windows_arch } h.arch \wedge \text{proto_of } sock.pr = \text{PROTO_UDP})) \wedge \\ & \quad (\forall tcp_sock. sock.pr = \text{TCP_PROTO}(tcp_sock) \implies \\ & \quad \quad tcp_sock.st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}; \text{LAST_ACK}; \text{FIN_WAIT_1}; \text{CLOSING}\} \vee \\ & \quad \quad (\neg sock.cantrcvmore \wedge tcp_sock.st = \text{FIN_WAIT_2}) \vee \\ & \quad \quad (\text{linux_arch } h.arch \wedge tcp_sock.st = \text{SYN_RECEIVED}) \vee \\ & \quad \quad \text{windows_arch } h.arch)) \end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a *getpeername*(*fd*) call is made where *fd* refers to a socket *sock* identified by *sid*. The socket does not have both its peer IP and port set, If it is a TCP socket then it is not in state ESTABLISHED, CLOSE_WAIT, LAST_ACK, FIN_WAIT_1 or CLOSING; or in state FIN_WAIT_2 and not shutdown for reading. The call fails with an ENOTCONN error.

A *tid*·*getpeername*(*fd*) transition is made, leaving the thread state RET(FAIL ENOTCONN).

Variations

| | |
|-------|--|
| Linux | As above, with the additional condition that if <i>sock</i> is a TCP socket then it is not in state SYN_RECEIVED. |
| WinXP | As above, except that if <i>sock</i> is a TCP socket then it does not matter what state it is in and if it is a UDP socket then the state of its peer port, whether it is set or unset, does not matter. |

15.11 getsockbopt() (TCP and UDP)

getsockbopt : (fd * sockbflag) → bool

A call to *getsockbopt*(*fd*, *flag*) returns the value of one of the socket's boolean-valued flags.

The *fd* argument is a file descriptor referring to the socket to retrieve a flag's value from, and the *flag* argument is the boolean-valued socket flag to get. Possible flags are:

- SO_BSDCOMPAT Reports whether the BSD semantics for delivery of ICMPs to UDP sockets with no peer address set is enabled.
- SO_DONTROUTE Reports whether outgoing messages bypass the standard routing facilities.
- SO_KEEPALIVE Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.
- SO_OOBINLINE Reports whether the socket leaves received out-of-band data (data marked urgent) inline.

- **SO_REUSEADDR** Reports whether the rules used in validating addresses supplied to `bind()` should allow reuse of local ports, if this is supported by the protocol.

The return value of the `getsockbopt()` call is the boolean-value of the specified socket flag.

15.11.1 Errors

A call to `getsockbopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------------|--|
| ENOPROTOOPT | The specified flag is not supported by the protocol. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.11.2 Common cases

getsockbopt_1; return_1

15.11.3 API

`getsockbopt()` is Posix `getsockopt()` for boolean-valued socket flags.

```
Posix:      int getsockopt(int socket, int level, int option_name,
                        void *restrict option_value,
                        socklen_t *restrict option_len);
FreeBSD:    int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
Linux:      int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
WinXP:      int getsockopt(SOCKET s, int level, int optname,
                        char* optval, int* optlen);
```

In the Posix interface:

- **socket** is the file descriptor of the socket on which to get the flag, corresponding to the `fd` argument of the model `getsockbopt()`.
- **level** is the protocol level at which the flag resides: `SOL_SOCKET` for the socket level options, and **option_name** is the flag to be retrieved. These two correspond to the *flag* argument to the model `getsockbopt()` where the possible values of **option_name** are limited to: `SO_BSDCOMPAT`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_OOINLINE`, and `SO_REUSEADDR`.
- **option_value** is a pointer to a location of size **option_len** to store the value retrieved by `getsockopt()`. These two correspond to the `bool` return type of the model `getsockbopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.11.4 Model details

The following errors are not modelled:

- **EFAULT** signifies the pointer passed as **option_value** was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- **EINVAL** signifies the **option_name** was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `getsockbopt()`.
- **WSAEINPROGRESS** is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.11.5 Summary

| | | |
|----------------------|--------------------------|---|
| <i>getsockbopt_1</i> | all: fast succeed | Successfully retrieve value of boolean socket flag |
| <i>getsockbopt_2</i> | udp: fast succeed | Fail with ENOPROTOOPT: option not valid on WinXP UDP socket |

15.11.6 Rules

getsockbopt_1 **all: fast succeed** Successfully retrieve value of boolean socket flag

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{getsockbopt}(fd, f)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(sf.b(f))))_{\text{sched_timer}}) \rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sf = (h.socks[sid]).sf \wedge$
 $(\text{windows_arch } h.arch \wedge \text{proto_of}(h.socks[sid]).pr = \text{PROTO_UDP})$
 $\implies f \notin \{\text{SO_KEEPALIVE}; \text{SO_OOBINLINE}\}$

Description

From thread *tid*, which is in the RUN state, a *getsockbopt*(*fd*, *f*) call is made. *fd* refers to a socket *sid* with boolean socket flags *sf.b*, and *f* is a boolean socket flag. The call succeeds, returning the value of *f*: **T** if *f* is set, and **F** if *f* is not set in *sf.b*.

A *tid*·*getsockbopt*(*fd*, *f*) transition is made, leaving the thread state *RET*(*OK*(*sf.b*(*f*))) where *sf.b*(*f*) is the boolean value of the socket's flag *f*.

Variations

| | |
|-------|--|
| WinXP | As above, except that if <i>sid</i> is a UDP socket, then <i>f</i> cannot be SO_KEEPALIVE or SO_OOBINLINE. |
|-------|--|

getsockbopt_2 **udp: fast succeed** Fail with ENOPROTOOPT: option not valid on WinXP UDP socket

$$\begin{aligned}
 &h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 &\quad socks := socks \oplus \\
 &\quad \quad [(sid, sock \langle pr := \text{UDP_PROTO}(udp) \rangle)] \rangle \\
 &\xrightarrow{tid \cdot \text{getsockbopt}(fd, f)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}); \\
 &\quad socks := socks \oplus \\
 &\quad \quad [(sid, sock \langle pr := \text{UDP_PROTO}(udp) \rangle)] \rangle
 \end{aligned}$$

$\text{windows_arch } h.arch \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $f \in \{\text{SO_KEEPALIVE}; \text{SO_OOBINLINE}\}$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the RUN state, a `getsockbopt(fd, f)` call is made, where *f* is either `SO_KEEPALIVE` or `SO_OOBINLINE`. The call fails with an `ENOPROTOOPT` error.

A *tid*·`getsockbopt(fd, f)` transition is made, leaving the thread state `RET(FAIL ENOPROTOOPT)`.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.12 getsockerr() (TCP and UDP)

`getsockerr` : `fd` → `unit`

A call `getsockerr(fd)` returns the pending error of a socket, clearing it, if there is one.

`fd` is a file descriptor referring to a socket. If the socket has a pending error then the `getsockerr()` call will fail with that error, otherwise it will return successfully.

15.12.1 Errors

In addition to failing with the pending error, a call to `getsockerr()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-----------------------|--|
| <code>EBADF</code> | The file descriptor passed is not a valid file descriptor. |
| <code>ENOTSOCK</code> | The file descriptor passed does not refer to a socket. |

15.12.2 Common cases

getsockerr_1; return_1

getsockerr_2; return_1

15.12.3 API

`getsockerr()` is Posix `getsockopt()` for the `SO_ERROR` socket option.

```
Posix:      int getsockopt(int socket, int level, int option_name,
                        void *restrict option_value,
                        socklen_t *restrict option_len);
FreeBSD:   int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
Linux:     int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
WinXP:     int getsockopt(SOCKET s, int level, int optname,
                        char* optval, int* optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to get the option on, corresponding to the `fd` argument of the model `getsockerr()`.
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. For `getsockerr()` `option_name` is set to `SO_ERROR`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsockopt()`. When `option_name` is `SO_ERROR` these fields are not used.

- the returned `int` is either 0 to indicate the socket has no pending error or -1 to indicate a pending error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.12.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, the flag for `getsockerr()` is always `SO_ERROR` so this error cannot occur.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.12.5 Summary

| | | |
|---------------------------|--------------------------|---|
| <code>getsockerr_1</code> | all: fast succeed | Return successfully: no pending error |
| <code>getsockerr_2</code> | all: fast fail | Fail with pending error and clear the error |

15.12.6 Rules

`getsockerr_1` **all: fast succeed** Return successfully: no pending error

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid \cdot \text{getsockerr}(fd)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}) \rangle \rangle$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &(h.socks[sid]).es = * \end{aligned}$$

Description

From thread `tid`, which is in the `RUN` state, a `getsockerr(fd)` call is made. `fd` refers to a socket `sid` which has no pending errors. The call succeeds.

A `tid.getsockerr(fd)` transition is made, leaving the thread state `RET(OK())`.

`getsockerr_2` **all: fast fail** Fail with pending error and clear the error

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, sock)] \rangle \rangle \xrightarrow{tid \cdot \text{getsockerr}(fd)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}); socks := socks \oplus [(sid, sock')] \rangle \rangle$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &\uparrow e = sock.es \wedge \\ &sock' = sock \langle \langle es := * \rangle \rangle \end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a `getsockerr(fd)` call is made. *fd* refers to a socket *sid* which has pending error *e*. The call fails, returning *e*.

A *tid*.`getsockerr(fd)` transition is made, leaving the thread state `RET(FAIL e)` and clearing the error *e* from the socket.

15.13 getsocklistening() (TCP and UDP)

`getsocklistening : fd → bool`

A call to `getsocklistening(fd)` returns **T** if the socket referenced by *fd* is listening, or **F** otherwise. For TCP a socket is listening if it is in the LISTEN state. For UDP, which is not a connection-oriented protocol, a socket can never be listening.

15.13.1 Errors

A call to `getsocklistening()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------------|---|
| ENOPROTOOPT | FreeBSD does not support this socket option, and on Linux and WinXP this option is not supported for UDP sockets. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.13.2 Common cases

getsocklistening_1; return_1

15.13.3 API

`getsocklistening()` is Posix `getsockopt()` for the `SO_ACCEPTCONN` socket option.

Posix: `int getsockopt(int socket, int level, int option_name, void *restrict option_value, socklen_t *restrict option_len);`

FreeBSD: `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);`

Linux: `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);`

WinXP: `int getsockopt(SOCKET s, int level, int optname, char* optval, int* optlen);`

In the Posix interface:

- `socket` is the file descriptor of the socket to get the option on, corresponding to the `fd` argument of the model `getsocklistening()`.
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. For `getsocklistening()` `option_name` is set to `SO_ACCEPTCONN`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsockopt()`. The value stored in the location corresponds to the `bool` return value of the model `getsocklistening()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The Linux and WinXP interfaces are similar except where noted. FreeBSD does not support the `SO_ACCEPTCONN` socket option.

15.13.4 Model details

The following errors are not modelled:

- EFAULT signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- EINVAL signifies the `option_name` was invalid at the specified socket `level`. In the model, the flag for `getsocklistening()` is always `SO_ACCEPTCONN` so this error cannot occur.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.13.5 Summary

getsocklistening_1 **tcp: fast succeed**

getsocklistening_3 **tcp: fast fail**

getsocklistening_2 **udp: rc**

Return successfully: **T** if socket is listening, **F** otherwise

Fail with `ENOPROTOOPT`: on FreeBSD operation not supported

Return **F** or fail with `ENOPROTOOPT`: a UDP socket cannot be listening

15.13.6 Rules

getsocklistening_1 **tcp: fast succeed** Return successfully: **T** if socket is listening, **F** otherwise

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid.\text{getsocklistening}(fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } b))_{\text{sched_timer}}) \rangle$$

$fd \in \text{dom}(h.fds) \wedge$

$fd = h.fds[fd] \wedge$

$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$

$\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge$

$b = (tcp_sock.st = \text{LISTEN}) \wedge$

$\neg(\text{bsd_arch } h.arch)$

Description

From thread *tid*, which is in the `RUN` state, a `getsocklistening(fd)` call is made where *fd* refers to a TCP socket *sid*.

A `tid.getsocklistening(fd)` transition is made, leaving the thread state `RET(OK b)` where $b = \mathbf{T}$ if the socket is in the `LISTEN` state, and $b = \mathbf{F}$ otherwise.

Variations

| | |
|---------|---|
| FreeBSD | This rule does not apply: see <i>getsocklistening_3</i> . |
|---------|---|

getsocklistening_3 **tcp: fast fail** Fail with `ENOPROTOOPT`: on FreeBSD operation not supported

$$\xrightarrow{tid.\text{getsocklistening}(fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } \text{ENOPROTOOPT}))_{\text{sched_timer}}) \rangle$$

$\text{bsd_arch } h.arch \wedge$

$$\begin{aligned}
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr
\end{aligned}$$

Description

On FreeBSD, a `getsocklistening(fd)` call is made from thread `tid` which is in the RUN state where `fd` refers to a TCP socket `sid`. The call fails with an ENOPROTOOPT error.

A `tid.getsocklistening(fd)` transition is made, leaving the thread state `RET(FAIL ENOPROTOOPT)`.

Variations

| | |
|-------|---|
| Linux | This rule does not apply: see <i>getsocklistening_1</i> . |
| WinXP | This rule does not apply: see <i>getsocklistening_1</i> . |

getsocklistening_2 **udp: rc** Return **F** or fail with ENOPROTOOPT: a UDP socket cannot be listening

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_a) \rangle \xrightarrow{tid.getsocklistening(fd)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched.timer}}) \rangle$$

$$\begin{aligned}
&\text{proto_of}(h.socks[sid]).pr = \text{PROTO_UDP} \wedge \\
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&\mathbf{if} \text{ linux_arch } h.arch \mathbf{then} rc = \text{FAST SUCCEED} \wedge ret = \text{OK } \mathbf{F} \\
&\mathbf{else} rc = \text{FAST FAIL} \wedge ret = \text{FAIL ENOPROTOOPT}
\end{aligned}$$

Description

Consider a UDP socket `sid`, referenced by `fd`. From thread `tid`, which is in the RUN state, a `getsocklistening(fd)` call is made. On Linux the call succeeds, returning **F**; on FreeBSD and WinXP the call fails with an ENOPROTOOPT error.

A `tid.getsocklistening(fd)` transition is made, leaving the thread state `RET(OK(F))` on Linux, and `RET(FAIL ENOPROTOOPT)` on FreeBSD and Linux.

Variations

| | |
|---------|---|
| Posix | As above: the call fails with an ENOPROTOOPT error. |
| FreeBSD | As above: the call fails with an ENOPROTOOPT error. |
| Linux | As above: the call succeeds, returning F . |
| WinXP | As above: the call fails with an ENOPROTOOPT error. |

15.14 getsockname() (TCP and UDP)

`getsockname` : `fd` \rightarrow (`ip option` * `port option`)

A call to `getsockname(fd)` returns the local address pair of a socket. If the file descriptor `fd` refers to the socket `sock` then the return value of a successful call will be $(sock.is_1, sock.ps_1)$.

15.14.1 Errors

A call to `getsockname()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|------------|---|
| ECONNRESET | On FreeBSD, TCP socket has its <code>cb.bsd_cantconnect</code> flag set due to previous connection establishment attempt. |
| EINVAL | Socket not bound to local address on WinXP. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| ENOBUFS | Out of resources. |

15.14.2 Common cases

getsockname_1; return_1

15.14.3 API

Posix: `int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`

FreeBSD: `int getsockname(int s, struct sockaddr *name, socklen_t *namelen);`

Linux: `int getsockname(int s, struct sockaddr *name, socklen_t *namelen);`

WinXP: `int getsockname(SOCKET s, struct sockaddr* name, int* namelen);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to get the local address of, corresponding to the `fd` argument in the model `getsockname()`.
- `address` is a pointer to a `sockaddr` structure of length `address_len`, which contains the local address of the socket upon return. These two correspond to the `(ip option, port option)` return type of the model `getsockname()`. If the `sin_addr.s_addr` field of the `name` structure is set to 0 on return, then the socket's local IP address is not set: the `ip option` member of the return tuple is set to `*`; otherwise, if it is set to `i` then it corresponds to the socket having local IP address and so the `ip option` member of the return tuple is $\uparrow i$. If the `sin_port` field of the `name` structure is set to 0 on return then the socket does not have a local port set, corresponding to the `port option` in the return tuple being `*`; otherwise the `sin_port` field is set to `p` corresponding to the socket having its local port set: the `port option` in the return tuple is $\uparrow p$.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.14.4 Model details

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, `EFAULT` can be returned if the `name` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `getsockname()` that is excluded by the clean interface used in the model `getsockname()`.
- in Posix, `EINVAL` can be returned if the socket has been shutdown. None of the implementations return `EINVAL` in this case.

- in Posix, EOPNOTSUPP is returned if the `getsockname()` operation is not supported by the protocol. Both UDP and TCP support this operation.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.14.5 Summary

| | | |
|----------------------|--------------------------|---|
| <i>getsockname_1</i> | all: fast succeed | Successfully return socket's local address |
| <i>getsockname_2</i> | tcp: fast fail | Fail with ECONNRESET: previous connection attempt has failed on FreeBSD |
| <i>getsockname_3</i> | all: fast fail | Fail with EINVAL: socket not bound on WinXP |

15.14.6 Rules

getsockname_1 **all: fast succeed** Successfully return socket's local address

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{getsockname}(fd) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(sock.is_1, sock.ps_1)))_{\text{sched_timer}}) \rangle}$$

$fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock = h.socks[sid] \wedge$
(case sock.pr of
TCP_PROTO(*tcp_sock*) \rightarrow
 $\text{bsd_arch } h.arch \implies \neg(\text{tcp_sock.cb.bsd_cantconnect} = \mathbf{T} \wedge \text{sock.ps}_1 = *) \parallel$
UDP_PROTO(*444*) $\rightarrow \mathbf{T} \wedge$
 $(\text{windows_arch } h.arch \implies \text{sock.is}_1 \neq * \vee \text{sock.ps}_1 \neq *)$
)

Description

From thread *tid*, which is in the RUN state, a `getsockname(fd)` call is made where *fd* refers to socket *sock*, identified by *sid*. The socket's local address is returned: $(sock.is_1, sock.ps_1)$.

A `tid.getsockname(fd)` transition is made, leaving the thread state $\text{RET}(\text{OK}(sock.is_1, sock.ps_1))$.

Variations

| | |
|---------|---|
| FreeBSD | This rule does not apply if the socket's <i>bsd_cantconnect</i> flag is set in its control block and its local port is not set. |
| WinXP | As above with the additional condition that either the socket's local IP address or local port must be set. |

getsockname_2 **tcp: fast fail** Fail with ECONNRESET: previous connection attempt has failed on FreeBSD

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$socks := socks \oplus [(sid, sock)] \rangle$$

$$\frac{tid \cdot \text{getsockname}(fd)}{\quad} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ECONNRESET}))_{\text{sched_timer}}); \text{socks} := \text{socks} \oplus [(sid, \text{sock})] \rangle$$

bsd_arch $h.arch \wedge$
 sock.pr = TCP_PROTO(tcp_sock) \wedge
 (tcp_sock.cb.bsd_cantconnect = **T** \wedge sock.ps₁ = *) \wedge

$fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$

Description

On FreeBSD, from thread tid , which is in the RUN state, a $\text{getsockname}(fd)$ call is made where fd refers to a TCP socket $sock$, identified by sid , which has its $bsd_cantconnect$ flag set and is not bound to a local port.

A $tid \cdot \text{getsockname}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ECONNRESET})$.

Variations

| | |
|-------|---------------------------|
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

getsockname_3 **all: fast fail Fail with EINVAL: socket not bound on WinXP**

$$\frac{tid \cdot \text{getsockname}(fd)}{\quad} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{socks} := \text{socks} \oplus [(sid, \text{sock} \langle is_1 := *; ps_1 := * \rangle)] \rangle$$

$$\frac{tid \cdot \text{getsockname}(fd)}{\quad} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}); \text{socks} := \text{socks} \oplus [(sid, \text{sock} \langle is_1 := *; ps_1 := * \rangle)] \rangle$$

windows_arch $h.arch \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$

Description

On WinXP, a $\text{getsockname}(fd)$ call is made from thread tid which is in the RUN state. fd refers to a socket sid which has neither its local IP address nor its local port set. The call fails with an EINVAL error.

A $tid \cdot \text{getsockname}(fd)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EINVAL})$.

Variations

| | |
|---------|---------------------------|
| Posix | This rule does not apply. |
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.15 getsocknopt() (TCP and UDP)

getsocknopt : (fd * socknflag) → int

A call to `getsocknopt(fd, flag)` returns the value of one of the socket's numeric flags. The `fd` argument is a file descriptor referring to the socket to retrieve a flag's value from. The `flag` argument is a numeric socket flag. Possible flags are:

- `SO_RCVBUF` Reports receive buffer size information.
- `SO_RCVLOWAT` Reports the minimum number of bytes to process for socket input operations.
- `SO_SNDBUF` Reports send buffer size information.
- `SO_SNDLOWAT` Reports the minimum number of bytes to process for socket output operations.

The return value of the `getsocknopt()` call is the numeric-value of the specified `flag`.

15.15.1 Errors

A call to `getsocknopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------------|--|
| ENOPROTOOPT | The specified flag is not supported by the protocol. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.15.2 Common cases

getsocknopt_1; return_1

15.15.3 API

`getsocknopt()` is Posix `getsockopt()` for numeric socket flags.

```
Posix:      int getsockopt(int socket, int level, int option_name,
                        void *restrict option_value,
                        socklen_t *restrict option_len);
FreeBSD:    int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
Linux:      int getsockopt(int s, int level, int optname,
                        void *optval, socklen_t *optlen);
WinXP:      int getsockopt(SOCKET s, int level, int optname,
                        char* optval, int* optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the `fd` argument of the model `getsocknopt()`.
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. These two correspond to the `flag` argument to the model `getsocknopt()` where the possible values of `option_name` are limited to `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF` and `SO_SNDLOWAT`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsocknopt()`. They correspond to the `int` return type of the model `getsocknopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.15.4 Model details

The following errors are not modelled:

- EFAULT signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error WSAEFAULT may also signify that the `optlen` parameter was too small.
- EINVAL signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `getsocknopt()`.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.15.5 Summary

| | | |
|----------------------------|--------------------------|---|
| <code>getsocknopt_1</code> | all: fast succeed | Successfully retrieve value of a numeric socket flag |
| <code>getsocknopt_4</code> | all: fast fail | Fail with ENOPROTOOPT: value of SO_RCVLOWAT and SO_SNDLOWAT not retrievable |

15.15.6 Rules

`getsocknopt_1` **all: fast succeed** Successfully retrieve value of a numeric socket flag

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{getsocknopt}(fd, f) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{int_of_num}(sf.n(f))))))_{\text{sched_timer}} \rangle}$$

$fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sf = (h.socks[sid]).sf \wedge$
 $(\text{windows_arch } h.arch \implies f \notin \{\text{SO_RCVLOWAT}; \text{SO_SNDLOWAT}\})$

Description

Consider the socket `sid`, referenced by `fd`, with socket flags `sf`. From thread `tid`, which is in the RUN state, a `getsocknopt(fd, f)` call is made. `f` is a numeric socket flag whose value is to be returned. The call succeeds, returning `sf.n(f)`, the numeric value of flag `f` for socket `sid`.

A `tid.getsocknopt(fd, f)` transition is made, leaving the thread state `RET(OK(int_of_num(sf.n(f)))`.

Variations

| | |
|-------|--|
| WinXP | The flag <code>f</code> is not SO_RCVLOWAT or SO_SNDLOWAT. |
|-------|--|

`getsocknopt_4` **all: fast fail** Fail with ENOPROTOOPT: value of SO_RCVLOWAT and SO_SNDLOWAT not retrievable

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{getsocknopt}(fd, f) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}) \rangle}$$

$\text{windows_arch } h.arch \wedge$
 $f \in \{\text{SO_RCVLOWAT}; \text{SO_SNDLOWAT}\}$

Description

From thread *tid*, which is in the RUN state, a `getsockopt(fd, f)` call is made where *fd* is a file descriptor. *f* is a numeric socket flag: either `SO_RCVLOWAT` or `SO_SNDLOWAT`, both flags whose value is non-retrievable. The call fails with an `ENOPROTOOPT` error.

A *tid*.`getsockopt(fd, f)` transition is made, leaving the thread state `RET(FAIL ENOPROTOOPT)`.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.16 getsockopt() (TCP and UDP)

`getsockopt : (fd * socktflag) → (int * int) option`

A call to `getsockopt(fd, flag)` returns the value of one of the socket's time-option flags.

The *fd* argument is a file descriptor referring to the socket to retrieve a flag's value from. The *flag* argument is a time option socket flag. Possible flags are:

- `SO_RCVTIMEO` Reports the timeout value for input operations.
- `SO_SNDTIMEO` Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent.

The return value of the `getsockopt()` call is the time-value of the specified *flag*. A return value of `*` means the timeout is disabled. A return value of $\uparrow(s, ns)$ means the timeout value is *s* seconds and *ns* nano-seconds.

15.16.1 Errors

A call to `getsockopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|--------------------------|--|
| <code>ENOPROTOOPT</code> | The specified flag is not supported by the protocol. |
| <code>EBADF</code> | The file descriptor passed is not a valid file descriptor. |
| <code>ENOTSOCK</code> | The file descriptor passed does not refer to a socket. |

15.16.2 Common cases

getsockopt_1; return_1

15.16.3 API

`getsockopt()` is Posix `getsockopt()` for time-valued socket options.

```

Posix:    int getsockopt(int socket, int level, int option_name,
                void *restrict option_value,
                socklen_t *restrict option_len);
FreeBSD:  int getsockopt(int s, int level, int optname,
                void *optval, socklen_t *optlen);
Linux:    int getsockopt(int s, int level, int optname,
                void *optval, socklen_t *optlen);
WinXP:    int getsockopt(SOCKET s, int level, int optname,
                char* optval, int* optlen);

```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the `fd` argument of the model `getsockopt()`.
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. These two correspond to the *flag* argument to the model `getsockopt()` where the possible values of `option_name` are limited to `SO_RCVTIMEO` and `SO_SNDTIMEO`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsockopt()`. They correspond to the `(int * int)` `option` return type of the model `getsockopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.16.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `getsockopt()`.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.16.5 Summary

| | | |
|---------------------------|--------------------------|--|
| <code>getsockopt_1</code> | all: fast succeed | Successfully retrieve value of time-option socket flag |
| <code>getsockopt_4</code> | all: fast fail | Fail with <code>ENOPROTOOPT</code> : on WinXP <code>SO_LINGER</code> not retrievable for UDP sockets |

15.16.6 Rules

`getsockopt_1` **all: fast succeed** Successfully retrieve value of time-option socket flag

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{getsockopt}(fd, f)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } t))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned}
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&sf = (h.socks[sid]).sf \wedge \\
&t = \text{tltimeopt_of_time}(sf.t(f)) \wedge
\end{aligned}$$

$$\neg(\text{windows_arch } h.\text{arch} \wedge \text{proto_of}(h.\text{socks}[sid]).\text{pr} = \text{PROTO_UDP} \wedge f = \text{SO_LINGER})$$

Description

From thread tid , which is in the RUN state, a `getsockopt(fd, f)` call is made. fd is a file descriptor referring to the socket sid which has socket flags sf , and f is a time-option flag. The call succeeds, returning $\text{OK}(t)$ where t is the value of the socket's flag f .

A $tid.\text{getsockopt}(fd, f)$ transition is made, leaving the thread state $\text{RET}(\text{OK}t)$.

Model details

The return type is $(\text{int} * \text{int})$ option, but the type of a time-option socket flag is `time`. The auxiliary function `tltimeopt_of_time` is used to do the conversion.

Variations

| | |
|-------|---|
| WinXP | As above but in addition if fd refers to a UDP socket then the flag is not <code>SO_LINGER</code> . |
|-------|---|

getsockopt_4 **all: fast fail** Fail with **ENOPROTOOPT**: on WinXP `SO_LINGER` not retrievable for UDP sockets

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$$

$$\underline{tid.\text{getsockopt}(fd, f)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}) \rangle$$

$$\text{windows_arch } h.\text{arch} \wedge$$

$$fd \in \text{dom}(h.\text{fds}) \wedge$$

$$fd = h.\text{fds}[fd] \wedge$$

$$h.\text{files}[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$\text{proto_of}(h.\text{socks}[sid]).\text{pr} = \text{PROTO_UDP} \wedge$$

$$f = \text{SO_LINGER}$$

Description

On WinXP, from thread tid which is in the RUN state, a `getsockopt(fd, f)` call is made. fd is a file descriptor referring to a UDP socket sid and f is the socket flag `SO_LINGER`. The flag f is not retrievable so the call fails with an `ENOPROTOOPT` error.

A $tid.\text{getsockopt}(fd, f)$ transition is made, leaving the thread state $\text{RET}(\text{ENOPROTOOPT})$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.17 listen() (TCP only)

`listen : fd * int → unit`

A call to `listen(fd, n)` puts a TCP socket that is in the `CLOSED` state into the `LISTEN` state, making it a passive socket, so that incoming connections for the socket will be accepted by the host and placed on its listen queue. Here fd is a file descriptor referring to the socket to put into the `LISTEN` state and n is

the *backlog* used to calculate the maximum lengths of the two components of the socket's listen queue: its pending connections queue, *lis.q₀*, and its complete connection queue, *lis.q*. The details of this calculation vary between architectures. The maximum useful value of *n* is SOMAXCONN: if *n* is greater than this then it will be truncated without generating an error. The minimum value of *n* is 0: if it a negative integer then it will be set to 0.

Once a socket is in the LISTEN state, `listen()` can be called again to change the backlog value.

15.17.1 Errors

A call to `listen()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|------------|---|
| EADDRINUSE | Another socket is listening on this local port. |
| EINVAL | On FreeBSD the socket has been shutdown for writing; on Linux the socket is not in the CLOSED or LISTEN state; or on WinXP the socket is not bound, |
| EISCONN | On WinXP the socket is already connected: it is not in the CLOSED or LISTEN state. |
| EOPNOTSUPP | The <code>listen()</code> operation is not supported for UDP. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.17.2 Common cases

A TCP socket is created, has its local address and port set by `bind()`, and then is put into the LISTEN state which can accept new incoming connections: *socket₁; return₁; bind₁ return₁; listen₁; return₁; ...*

15.17.3 API

Posix: `int listen(int socket, int backlog);`

FreeBSD: `int listen(int s, int backlog);`

Linux: `int listen(int s, int backlog);`

WinXP: `int listen(SOCKET s, int backlog);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to put into the LISTEN state, corresponding to the `fd` argument of the model `listen()`.
- `backlog` is an `int` on which the maximum permitted length of the socket's listen queue depends. It corresponds to the *n* argument of the model `listen()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.17.4 Model details

The following errors are not modelled:

- In Posix, `EACCES` may be returned if the calling process does not have the appropriate privileges. This is not modelled here.
- In Posix, `EDESTADDRREQ` shall be returned if the socket is not bound to a local address and the protocol does not support listening on an unbound socket. WinXP returns an `EINVAL` error in this case; FreeBSD and Linux autobind the socket if `listen()` is called on an unbound socket.

- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.17.5 Summary

| | | |
|------------------|--------------------------|---|
| <i>listen_1</i> | tcp: fast succeed | Successfully put socket in LISTEN state |
| <i>listen_1b</i> | tcp: fast succeed | Successfully update backlog value |
| <i>listen_1c</i> | tcp: fast succeed | Successfully put socket in the LISTEN state from any non- {CLOSED;LISTEN} state on FreeBSD |
| <i>listen_2</i> | tcp: fast fail | Fail with EINVAL on WinXP: socket not bound to local port |
| <i>listen_3</i> | tcp: fast fail | Fail with EINVAL on Linux or EISCONN on WinXP: socket not in CLOSED or LISTEN state |
| <i>listen_4</i> | tcp: fast fail | Fail with EADDRINUSE on Linux: another socket already listening on local port |
| <i>listen_5</i> | tcp: fast fail | Fail with EINVAL on BSD: socket shutdown for writing or <i>bsd_cantconnect</i> flag set |
| <i>listen_7</i> | udp: fast fail | Fail with EOPNOTSUPP: listen() called on UDP socket |

15.17.6 Rules

listen_1 **tcp: fast succeed** Successfully put socket in LISTEN state

$$\begin{aligned}
 & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 & \text{socks} := \text{socks} \oplus \\
 & \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, \text{cantrecvmore}, \\
 & \quad \quad \quad \text{TCP_Sock}(\text{CLOSED}, cb, *, [], *, [], *, \text{NO_OOBDATA}))]); \\
 & \text{listen} := \text{listen}_0 \rangle \\
 \hline
 & \text{tid} \cdot \text{listen}(fd, n) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
 & \text{socks} := \text{socks} \oplus \\
 & \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, \mathbf{F}, \text{cantrecvmore}, \\
 & \quad \quad \quad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, [], *, [], *, \text{NO_OOBDATA}))]); \\
 & \text{listen} := sid :: \text{listen}_0; \\
 & \text{bound} := \text{bound} \rangle
 \end{aligned}$$

$$\begin{aligned}
 & fd \in \mathbf{dom}(h.fds) \wedge \\
 & fid = h.fds[fd] \wedge \\
 & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 & (\text{bsd_arch } h.arch \vee \text{cantrecvmore} = \mathbf{F}) \wedge \\
 & \neg(\text{windows_arch } h.arch \wedge \text{IS_NONE } ps_1) \wedge \\
 & (\text{bsd_arch } h.arch \implies cb.\text{bsd_cantconnect} = \mathbf{F}) \wedge \\
 & p_1 \in \text{autobind}(ps_1, \text{PROTO_TCP}, \text{socks} \setminus \{sid\}) \wedge \\
 & (\text{if } ps_1 = * \text{ then } \text{bound} = sid :: h.\text{bound} \text{ else } \text{bound} = h.\text{bound}) \wedge \\
 & lis = \langle q_0 := []; \\
 & \quad q := []; \\
 & \quad qlimit := n \rangle
 \end{aligned}$$

Description

From thread *tid*, which is currently in the RUN state, a `listen(fd, n)` call is made. *fd* is a file descriptor referring to a TCP socket identified by *sid* which is not shutdown for writing, is in the CLOSED state, has an empty send and receive queue, and does not have its send or receive urgent pointers set. The host's list of listening sockets is `listen0`. Either the socket is bound to a local port *p*₁, or it can be autobound to a local port *p*₁.

The call succeeds: a tid -listen(fd, n) transition is made, leaving the thread in state $\text{RET}(\text{OK}())$. The socket is put in the LISTEN state, with an empty listen queue, lis , with n as its backlog. sid is added to the host's list of listening sockets, $\text{listen} := sid :: \text{listen}_0$, and if autobinding occurred, it is also added to the host's list of bound sockets, $h.\text{bound}$, to create a new list bound .

Variations

| | |
|---------|--|
| FreeBSD | The <code>bsd_cantconnect</code> flag in the control block must not be set to T (from an earlier connection establishment attempt). |
| WinXP | As above, except that the socket must be bound to a local port p_1 . If it is not bound then autobinding will not occur: the call will fail with an EINVAL error. See also <i>listen_2</i> (p195). |

listen_1b **tcp: fast succeed** Successfully update backlog value

$$\begin{array}{l} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ socks := socks \oplus \\ \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, \text{cantrcvmore}, \\ \quad \quad \quad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, [], *, [], *, \text{NO_OOBDATA})))]); \\ \text{listen} := \text{listen}_0 \rangle \\ \hline \text{tid}\cdot\text{listen}(fd, n) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\ socks := socks \oplus \\ \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, \text{cantrcvmore}, \\ \quad \quad \quad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis', [], *, [], *, \text{NO_OOBDATA})))]); \\ \text{listen} := sid :: \text{listen}_0 \rangle \end{array}$$

$$\begin{array}{l} fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ (\text{bsd_arch } h.arch \vee \text{cantrcvmore} = \mathbf{F}) \wedge \\ lis' = lis \langle qlimit := n \rangle \end{array}$$

Description

From thread tid , which is in the RUN state, a $\text{listen}(fd, n)$ call is made. fd refers to a TCP socket identified by sid which is currently in the LISTEN state. The host has a list of listening sockets, listen_0 . The call succeeds.

A tid -listen(fd, n) transition is made, leaving the thread state $\text{RET}(\text{OK}())$. The backlog value of the socket's listen queue, $lis.qlimit$ is updated to be n , resulting in a new listen queue lis' for the socket. sid is added to the head of the host's listen queue, $\text{listen} := sid :: \text{listen}_0$.

listen_1c **tcp: fast succeed** Successfully put socket in the LISTEN state from any non-**{CLOSED; LISTEN}** state on FreeBSD

$$\begin{array}{l} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ socks := socks \oplus \\ \quad [(sid, sock)]; \\ \text{listen} := \text{listen}_0 \rangle \\ \hline \text{tid}\cdot\text{listen}(fd, n) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\ socks := socks \oplus \\ \quad [(sid, sock')]; \\ \text{listen} := sid :: \text{listen}_0 \rangle \end{array}$$

$$\begin{array}{l} \text{bsd_arch } h.arch \wedge \\ fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \end{array}$$

$$\begin{aligned}
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&sock = \text{SOCK}(\uparrow fd, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \text{TCP_PROTO}(tcp_sock)) \wedge \\
&tcp_sock.st \notin \{\text{CLOSED}; \text{LISTEN}\} \wedge \\
&sock' = sock \langle pr := \text{TCP_PROTO}(tcp_sock \langle st := \text{LISTEN}; lis := \uparrow lis \rangle) \rangle \wedge \\
&lis = \langle q_0 := []; \\
&\quad q := []; \\
&\quad qlimit := n \rangle
\end{aligned}$$

Description

On BSD, calling `listen()` always succeeds on a socket regardless of its state: the state of the socket is just changed to `LISTEN`.

From thread `tid`, which is in the `RUN` state, a `listen(fd, n)` call is made. `fd` refers to a TCP socket identified by `sid` which is currently in any non-`{CLOSED; LISTEN}` state. The call succeeds.

A `tid·listen(fd, n)` transition is made, leaving the thread state `RET(OK())`. The socket state is updated to `LISTEN`, with empty listen queues.

listen_2 **tcp: fast fail** Fail with `EINVAL` on WinXP: socket not bound to local port

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{listen}(fd, n)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } \text{EINVAL}))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned}
&\text{windows_arch } h.arch \wedge \\
&fd \in \text{dom}(h.fds) \wedge \\
&fd = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&h.socks[sid] = sock \wedge \\
&\text{proto_of } sock.pr = \text{PROTO_TCP} \wedge \\
&sock.ps_1 = *
\end{aligned}$$

Description

On WinXP, from thread `tid`, which is in the `RUN` state, a `listen(fd, n)` call is made. `fd` refers to a TCP socket `sock`, identified by `sid`, which is not bound to a local port: `sock.ps_1 = *`. The call fails with an `EINVAL` error.

A `tid·listen(fd, n)` transition is made, leaving the thread state `RET(FAIL EINVAL)`.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

listen_3 **tcp: fast fail** Fail with `EINVAL` on Linux or `EISCONN` on WinXP: socket not in `CLOSED` or `LISTEN` state

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \xrightarrow{tid \cdot \text{listen}(fd, n)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned}
&fd \in \text{dom}(h.fds) \wedge \\
&fd = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&h.socks[sid] = sock \wedge \\
&sock.pr = \text{TCP_PROTO}(tcp_sock) \wedge \\
&tcp_sock.st \notin \{\text{CLOSED}; \text{LISTEN}\} \wedge
\end{aligned}$$

```

¬(bsd_arch h.arch) ∧
(if windows_arch h.arch then
  err = EISCONN
else if linux_arch h.arch then
  err = EINVAL
else
  F)

```

Description

From thread tid , which is in the RUN state, a $\text{listen}(fd, n)$ call is made. fd refers to a TCP socket $sock$, identified by sid , which is not in the CLOSED or LISTEN state. On Linux the call fails with an EINVAL error; on WinXP it fails with an EISCONN error.

A $tid.\text{listen}(fd, n)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } err)$ where err is one of the above errors.

Variations

| | |
|---------|---|
| FreeBSD | This rule does not apply: $\text{listen}()$ can be called from any state. |
| Linux | As above: the call fails with an EINVAL error. |
| WinXP | As above: the call fails with an EISCONN error. |

listen_4 **tcp: fast fail** Fail with EADDRINUSE on Linux: another socket already listening on local port

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid.\text{listen}(fd, n)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } \text{EADDRINUSE}))_{\text{sched_timer}}) \rangle$$

```

linux_arch h.arch ∧
fd ∈ dom(h.fds) ∧
fd = h.fds[fd] ∧
h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
h.socks[sid] = sock ∧
sock.pr = TCP_PROTO(tcp_sock) ∧
tcp_sock.st = CLOSED ∧
sock.ps1 = ↑ p1 ∧
(∃ sid' sock' tcp_sock'.h.socks[sid'] = sock' ∧ sock'.pr = TCP_PROTO(tcp_sock') ∧
  tcp_sock'.st = LISTEN ∧ sock'.ps1 = sock.ps1 ∧
  ¬(∃ i1 i'1.i1 ≠ i'1 ∧ sock.is1 = ↑ i1 ∧ sock'.is1 = ↑ i'1))

```

Description

On Linux, from thread tid , which is in the RUN state, a $\text{listen}(fd, n)$ call is made. fd refers to a TCP socket $sock$, identified by sid , in state CLOSED and bound to local port p_1 . There is another TCP socket, $sock'$, in the host's finite map of sockets, $h.socks$ that is also bound to local port p_1 , and is in the LISTEN state. The two sockets, $sock$ and $sock'$, are not bound to different IP addresses: either they are both bound to the same IP address, one is bound to an IP address and the other is not bound to an IP address, or neither is bound to an IP address. The call fails with an EADDRINUSE error.

A $tid.\text{listen}(fd, n)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL } \text{EADDRINUSE})$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| WinXP | This rule does not apply. |

listen_5 **tcp: fast fail** Fail with EINVAL on BSD: socket shutdown for writing or *bsd_cantconnect* flag set

$$\begin{aligned}
 & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle; \\
 & \text{socks} := \text{socks} \oplus \\
 & \quad \langle (sid, \text{sock} \langle \text{cantsndmore} := \text{cantsndmore}; pr := \text{TCP_PROTO}(\text{tcp_sock} \langle st := st \rangle) \rangle) \rangle \\
 \hline
 & \text{tid} \cdot \text{listen}(fd, n) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}) \rangle; \\
 & \quad \text{socks} := \text{socks} \oplus \\
 & \quad \quad \langle (sid, \text{sock} \langle \text{cantsndmore} := \text{cantsndmore}; pr := \text{TCP_PROTO}(\text{tcp_sock} \langle st := st \rangle) \rangle) \rangle
 \end{aligned}$$

$bsd_arch \ h.arch \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $st \in \{\text{CLOSED}; \text{LISTEN}\} \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $(\text{cantsndmore} = \mathbf{T} \vee \text{tcp_sock}.cb.bsd_cantconnect = \mathbf{T})$

Description

On FreeBSD, from thread *tid*, which is in the RUN state, a *listen*(*fd*, *n*) call is made. *fd* refers to a TCP socket *sock*, identified by *sid*, which is in the CLOSED or LISTEN state. The socket is either shutdown for writing or has its *bsd_cantconnect* flag set due to an earlier connection-establishment attempt. The call fails with an EINVAL error.

A *tid*·*listen*(*fd*, *n*) transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

| | |
|-------|---------------------------|
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

listen_7 **udp: fast fail** Fail with EOPNOTSUPP: listen() called on UDP socket

$$\begin{aligned}
 & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \\
 \hline
 & \text{tid} \cdot \text{listen}(fd, n) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EOPNOTSUPP}))_{\text{sched_timer}}) \rangle
 \end{aligned}$$

$fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $\text{proto_of}(h.socks[sid]).pr = \text{PROTO_UDP}$

Description

Consider a UDP socket *sid*, referenced by *fd*. From thread *tid*, which is in the RUN state, a *listen*(*fd*, *n*) call is made. The call fails with an EOPNOTSUPP error.

A *tid*·listen(*fd*, *n*) transition is made, leaving the thread state RET(FAIL EOPNOTSUPP).

Calling listen() on a socket for a connectionless protocol (such as UDP) is meaningless and is thus an unsupported (EOPNOTSUPP) operation.

15.18 pselect() (TCP and UDP)

pselect : (fd list * fd list * fd list * (int * int) option * signal list option) → (fd list * (fd list * fd list))

A call to pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) waits for one of the file descriptors in *readfds* to be ready for reading, *writefds* to be ready for writing, *exceptfds* to have a pending error, or for *timeout* to expire.

The *readfds* argument is a set of file descriptors to be checked for being ready to read. Broadly, a file descriptor *fd* is ready for reading if a *recv*(*fd*, *_, _*) call on the socket would not block, i.e. if there is data present or a pending error.

The *writefds* argument is a set of file descriptors to be checked for being ready to write. Broadly, a file descriptor *fd* is ready for writing if a *send*(*fd*, *_, _*, *_*) call would not block.

The *exceptfds* argument is a set of file descriptors to be checked for exception conditions pending. A file descriptor *fd* has an exception condition pending if there exists out-of-band data for the socket it refers to or the socket is still at the out-of-band mark.

The *timeout* argument specifies how long the pselect() call should block waiting for a file descriptor to be ready. If *timeout* = * then the call should block until one of the file descriptors in the *readfds*, *writefds*, or *exceptfds* becomes ready. If *timeout* = ↑(*s*, *ns*) then the call should block for at most *s* seconds and *ns* nanoseconds. However, system activity can lengthen the timeout interval by an indeterminate amount.

The *sigmask* argument is used to set the signal mask, the set of signals to be blocked. In the implementations, if *sigmask* = ↑(*siglist*) then pselect() first replaces the current signal mask by *siglist* before proceeding with the call, and then restores the original signal mask upon return. This specification does not model the dynamic behaviour of signals, however, and so we specify the behaviour of pselect() only for an empty signal mask.

A return value of (*readfds'*, (*writefds'*, *exceptfds'*)) from a pselect() call signifies that: the file descriptors in *readfds'* are ready for reading; the file descriptors in *writefds'* are reading for writing; and the file descriptors in *exceptfds'* have exceptional conditions pending.

If a pselect([], [], [], *Some*(*s*, *ns*), *sigmask*) call is made then the call will block for *s* seconds and *ns* nanoseconds or until a signal occurs.

To perform a poll, a pselect(*readfds*, *writefds*, *exceptfds*, *Some*(0, 0), *sigmask*) call should be made.

15.18.1 Errors

A call to pselect() can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------|---|
| EBADF | One or more of the file descriptors in a set is not a valid file descriptor. |
| EINVAL | Time-out not well-formed, file descriptor out of range, or on WinXP all file descriptor sets are empty. |
| ENOTSOCK | One or more of the file descriptors in a set is not a valid socket. |
| EINTR | The system was interrupted by a caught signal. |

15.18.2 Common cases

pselect() is called and returns immediately: *pselect_1*; *return_1*

pselect() blocks and then times out before any of the file descriptors become ready: *pselect_2*; *pselect_3*; *return_1*

pselect() blocks, TCP data is received from the network and processed, making a file descriptor ready for reading, and then pselect() returns: *pselect_1; deliver_in_99; deliver_in_3; pselect_2; return_1*

pselect() blocks, UDP data is received from the network and processed, making a file descriptor ready for reading, and then pselect() returns: *pselect_1; deliver_in_99; deliver_in_udp_1; pselect_2; return_1*

pselect() blocks, TCP data is sent to the network, an acknowledgement is received and processed, making a file descriptor ready for writing, and then pselect() returns: *pselect_1; deliver_out_1; deliver_out_99; deliver_in_99; deliver_in_3; pselect_2; return_1*

15.18.3 API

```
Posix:      int pselect(int nfd, fd_set *restrict readfds,
                    fd_set *restrict writefds, fd_set *restrict errorfds,
                    const struct timespec *restrict timeout,
                    const sigset_t *restrict sigmask);
FreeBSD:    int select(int nfd, fd_set *readfds, fd_set *writefds,
                    fd_set *exceptfds, struct timeval *timeout);
Linux:      int pselect(int n, fd_set *readfds, fd_set *writefds,
                    fd_set *exceptfds, const struct timespec *timeout,
                    const sigset_t *sigmask);
WinXP:      int select(int nfd, fd_set* readfds, fd_set* writefds,
                    fd_set* exceptfds, const struct timeval* timeout);
```

In the Posix interface:

- **nfd** specifies the range of file descriptors to be tested. The first **nfd** file descriptors shall be checked in each set. This is not necessary in the model pselect() as the file descriptor sets are implemented as a list rather than the integer arrays in Posix pselect().
- **readfds** on input specifies the file descriptors to be checked for being ready to read, corresponding to the *readfds* argument of the model pselect(). On output **readfds** indicates which of the file descriptors specified on input are ready to read, corresponding to the first **fd** list in the return type of the model pselect(). An **fd_set** is an integer array, where each bit of each integer corresponds to a file descriptor. If that bit is set then that file descriptor should be checked. **FD_CLR()**, **FD_ISSET()**, **FD_SET()**, and **FD_ZERO()** are provided to set bits in an **fd_set**.
- **writefds** on input specifies the file descriptors to be checked for being ready to write, corresponding to the *writefds* argument of the model pselect(). On output **writefds** indicates which of the file descriptors specified on input are ready to write, corresponding to the second **fd** list in the return type of the model pselect().
- **errorfds** on input specifies the file descriptors to be checked for pending error conditions, corresponding to the *exceptfds* argument of the model pselect(). On output **exceptfds** indicated which of the file descriptors specified on input have pending error conditions, corresponding to the third **fd** list in the return type of the model pselect().
- **timeout** specifies how long the pselect() call shall block before timing out, corresponding to the *timeout* argument of the model pselect(). If the **timeout** parameter is a null pointer this corresponds to *timeout = **; if the **timeout** parameter is not a null pointer, then its two fields, **timeout.tv_sec** (the number of seconds) and **timeout.tv_nsec** (the number of nano-seconds), correspond to *timeout = ↑(s, ns)* where *s* is the number of seconds, and *ns* is the number of nano-seconds.
- **sigmask** is the signal-mask to be used when examining the file descriptors, corresponding to the *sigmask* argument of the model pselect(). If **sigmask** is a null pointer then *sigmask = ** in the model; if **sigmask** is not a null pointer then *sigmask = ↑ sigs* in the model where *sigs* is the signal-mask to use.
- if the call is successful then the returned **int** is the number of bits set in the three **fd_set** arguments: the total number of file descriptors ready for reading, writing, or having exceptional conditions pending. Otherwise, the returned **int** is -1 to indicate an error, in which case the error code is in **errno**. On WinXP an error is indicated by a return value of **SOCKET_ERROR**, not -1, with the actual error code available through a call to **WSAGetLastError()**.

The Linux interface is similar. On FreeBSD and WinXP there is no pselect() call, only a select() call which is the same as the interface described above, except without the **sigmask** argument. The select() call

corresponds to calling the model `pselect()` with `sigmask = *`. Additionally, the `timeout` argument is a pointer to a `timeval` structure which has two members `tv_sec` and `tv_usec`, specifying the seconds and micro-seconds to block for, rather than seconds and nano-seconds.

The FreeBSD man page for `select()` warns of the following bug: "Version 2 of the Single UNIX Specification ("SUSv2") allows systems to modify the original timeout in place. Thus, it is unwise to assume that the timeout value will be unmodified by the `select()` call."

15.18.4 Model details

If the `pselect()` call blocks then the thread enters state `PSELECT2(readfds, writefds, exceptfds)` where:

- `readfds` : fd list is the list of file descriptors to be checked for being ready to read.
- `writefds` : fd list is the list of file descriptors to be checked for being ready to write.
- `exceptfds` : fd list is the list of file descriptors to be checked for pending exceptional conditions.

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.18.5 Summary

| | | |
|----------------------|------------------------------------|---|
| <i>pselect_1</i> | all: fast succeed | One or more file descriptors immediately ready, or no timeout set |
| <i>soreadable</i> | | check whether a socket is readable |
| <i>sowriteable</i> | | check whether a socket is writable |
| <i>soexceptional</i> | | check whether a socket is exceptional |
| <i>pselect_2</i> | all: block | Normal case |
| <i>pselect_3</i> | all: slow nonurgent succeed | Something becomes ready or pselect times out |
| <i>pselect_4</i> | all: fast fail | Fail with <code>EINVAL</code> : Timeout not well-formed |
| <i>pselect_5</i> | all: fast fail | Fail with <code>EINVAL</code> : File descriptor out of range |
| <i>pselect_6</i> | all: fast fail | Fail with <code>EBADF</code> or <code>ENOTSOCK</code> : Bad file descriptor |

15.18.6 Rules

| | | |
|--|--------------------------|---|
| <i>pselect_1</i> | all: fast succeed | One or more file descriptors immediately ready, or no timeout set |
| $h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$ | | |
| $\frac{tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}{h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(readfds'', writefds'', exceptfds''))_{\text{sched_timer}})) \rangle}$ | | |
| $(\text{ttimeout_wf } timeout \vee \text{windows_arch } h.\text{arch}) \wedge$ | | |
| $sigmask = * \wedge$ | | |
| $\neg(\exists fd \ n. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$ | | |
| $\text{if } \text{windows_arch } h.\text{arch}$ | | |
| $\text{then } n = (\max(\text{length } readfds)(\max(\text{length } writefds)(\text{length } exceptfds))) \wedge$ | | |
| $n \geq (\text{FD_SETSIZE } h.\text{arch})$ | | |
| else | | |
| $fd = \text{FD } n \wedge$ | | |
| $n \geq \text{FD_SETSIZE } h.\text{arch}) \wedge$ | | |
| $\text{badreadfds} = \text{filter}(\lambda fd.f d \notin \text{dom}(h.fds))readfds \wedge$ | | |
| $\text{badwritefds} = \text{filter}(\lambda fd.f d \notin \text{dom}(h.fds))writefds \wedge$ | | |

```

badexceptfds = filter( $\lambda fd. fd \notin \mathbf{dom}(h.fds)$ ) exceptfds  $\wedge$ 
(bsd_arch h.arch  $\vee$ 
(badreadfds = []  $\wedge$  badwritefds = []  $\wedge$  badexceptfds = []))  $\wedge$ 
 $\neg(\exists fd. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$ 
     $fd \notin \mathbf{dom}(h.fds)) \wedge$ 
readfds' = filter( $\lambda fd. \exists fid ff sid sock.$ 
     $fd \in \mathbf{dom}(h.fds) \wedge$ 
     $fid = h.fds[fid] \wedge$ 
     $h.files[fid] = \mathbf{FILE}(\mathbf{FT\_SOCKET}(sid), ff) \wedge$ 
     $sock = h.socks[sid] \wedge$ 
    soreadable h.arch sock) readfds  $\wedge$ 
writefds' = filter( $\lambda fd. \exists fid ff sid sock.$ 
     $fd \in \mathbf{dom}(h.fds) \wedge$ 
     $fid = h.fds[fid] \wedge$ 
     $h.files[fid] = \mathbf{FILE}(\mathbf{FT\_SOCKET}(sid), ff) \wedge$ 
     $sock = h.socks[sid] \wedge$ 
    sowriteable h.arch sock) writefds  $\wedge$ 
exceptfds' = filter( $\lambda fd. \exists fid ff sid sock.$ 
     $fd \in \mathbf{dom}(h.fds) \wedge$ 
     $fid = h.fds[fid] \wedge$ 
     $h.files[fid] = \mathbf{FILE}(\mathbf{FT\_SOCKET}(sid), ff) \wedge$ 
     $sock = h.socks[sid] \wedge$ 
    soexceptional h.arch sock) exceptfds  $\wedge$ 
(windows_arch h.arch  $\implies$  readfds  $\neq$  []  $\wedge$  writefds  $\neq$  []  $\wedge$  exceptfds  $\neq$  [])  $\wedge$ 
(readfds'  $\neq$  []  $\vee$  writefds'  $\neq$  []  $\vee$  exceptfds'  $\neq$  []  $\vee$  timeout =  $\uparrow(0, 0)$ )  $\wedge$ 
if windows_arch h.arch then
    readfds'' = readfds'  $\wedge$  writefds'' = writefds'  $\wedge$  exceptfds'' = exceptfds'
else
    readfds'' = INSERT_ORDERED readfds' readfds badreadfds  $\wedge$ 
    writefds'' = INSERT_ORDERED writefds' writefds badwritefds  $\wedge$ 
    exceptfds'' = INSERT_ORDERED exceptfds' exceptfds badexceptfds

```

Description

From thread *tid*, which is in the RUN state, a `pselect(readfds, writefds, exceptfds, timeout, sigmask)` call is made. The time-out is well-formed and no signal mask was set: *sigmask* = *. All of the file descriptors in the sets *readfds*, *writefds*, and *exceptfds* are greater than the maximum allowed file descriptor in a set for the architecture, FD_SETSIZE, and all of them are valid file descriptors: they are in the host's finite map of file descriptors, *h.fds*.

The call returns, without blocking, three sets: *readfds''*, *writefds''*, and *exceptfds''*. *readfds''* is the set of valid file descriptors in *readfds* that are ready for reading: a blocking `recv(fd, -, -)` call would not block; see soreadable (p202) for details. *writefds''* is the set of valid file descriptors in *writefds* that are ready for writing: a blocking `send(fd, -, -)` call would not block; see sowriteable (p202) for details. *exceptfds''* is the set of valid file descriptors in *exceptfds* that have pending exceptional conditions; see soexceptional (p203) for details.

One of these three sets must be non-empty or else a zero timeout was specified, *timeout* = $\uparrow(0, 0)$. A *tid*.`pselect(readfds, writefds, exceptfds, timeout, sigmask)` transition is made, leaving the thread state `RET(OK(readfds'', writefds'', exceptfds''))`.

Variations

| | |
|---------|---|
| FreeBSD | Invalid file descriptors (ones not in the host's finite map of file descriptors, <i>h.fds</i>) may be present in the sets <i>readfds</i> , <i>writefds</i> , and <i>exceptfds</i> , and all such file descriptors will then be included in the return sets <i>readfds''</i> , <i>writefds''</i> , and <i>exceptfds''</i> . |
| WinXP | On WinXP FD_SETSIZE is the maximum number of file descriptors in a set, so none of the sets <i>readfds</i> , <i>writefds</i> , and <i>exceptfds</i> has more than FD_SETSIZE members. Additionally, all three sets may not be empty. The time-out need not be well-formed because one or more file descriptors is immediately ready. |

– **check whether a socket is readable :**

```
soreadable arch sock =
case sock.pr of
TCP_PROTO(tcp) →
  (length tcp.rcvq ≥ sock.sf.n(SO_RCVLOWAT) ∨
   sock.cantrcvmore ∨
   (linux_arch arch ∧ tcp.st = CLOSED) ∨
   (tcp.st = LISTEN ∧
    ∃lis.tcp.lis = ↑ lis ∧
     lis.q ≠ [])) ∨
  sock.es ≠ *) ||
UDP_PROTO(udp) →
  (udp.rcvq ≠ [] ∨ sock.es ≠ * ∨ (sock.cantrcvmore ∧ ¬ windows_arch arch))
```

Description

A TCP socket *sock* is readable if: (1) the length of its receive queue is greater than or equal to the minimum number of bytes for socket input operations, *sf.n(SO_RCVLOWAT)*; (2) it has been shut down for reading; (3) on Linux, it is in the CLOSED state; it is in the LISTEN state and has at least one connection on its completed connection queue; or (4) it has a pending error.

A UDP socket *sock* is readable if its receive queue is not empty, it has a pending error, or it has been shutdown for reading.

Variations

| | |
|-------|--|
| Linux | On all OSes, attempting to read from a closed socket yields an immediate error. Only on Linux, however, does <i>soreadable</i> return T in this case. |
| WinXP | The socket will not be readable if it has been shutdown for reading. |

– **check whether a socket is writable :**

```
sowriteable arch sock =
case sock.pr of
TCP_PROTO(tcp) →
  ((tcp.st ∈ {ESTABLISHED; CLOSE_WAIT}) ∧
   sock.sf.n(SO_SNDBUF) – length tcp.sndq ≥ sock.sf.n(SO_SNDLOWAT)) ∨ (* change to send_buffer_space *)
  (if linux_arch arch then ¬sock.cantsndmore else sock.cantsndmore) ∨
  (linux_arch arch ∧ tcp.st = CLOSED) ∨
  sock.es ≠ *) ||
UDP_PROTO(udp) → T
```

Variations

| | |
|-------|--|
| Linux | On all OSes, attempting to write to a closed socket yields an immediate error. Only on Linux, however, does <i>sowriteable</i> return T in this case. On Linux, if the outgoing half of the connection has been closed by the application, the socket becomes non-writable, whereas on other OSes it becomes writable (because an immediate error would result from writing). |
|-------|--|

– **check whether a socket is exceptional :**

```
soexceptional arch sock =
case sock.pr of
TCP_PROTO(tcp) →
  (tcp.st = ESTABLISHED ∧
   (tcp.rcvurp = ↑ 0 ∨
    (∃c.tcp.iobc = OOBDATA c))) ||
UDP_PROTO(udp) → F
```

Description

A TCP socket has a pending exceptional condition if it is in state ESTABLISHED and has a pending byte of out-of-band data.

A UDP socket never has a pending exceptional condition.

pselect_2 **all: block** Normal case

$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$

$\underline{tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}$

$h \langle ts := ts \oplus (tid \mapsto (\text{PSELECT2}(readfds, writefds, exceptfds)_{\text{kern.timer } d'}) \rangle$

$\text{tlopt_wf } timeout \wedge$

$d' = \mathbf{min}(\text{time_of_tlopt } timeout) \text{ pselect_timeo_t_max} \wedge$

$sigmask = * \wedge$

$\neg(\exists fd n. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$

if windows_arch $h.arch$

then $n = \mathbf{max}(\mathbf{length } readfds)(\mathbf{max}(\mathbf{length } writefds)(\mathbf{length } exceptfds)) \wedge$

$n \geq \text{FD_SETSIZE } h.arch$

else

$fd = \text{FD } n \wedge$

$n \geq \text{FD_SETSIZE } h.arch) \wedge$

$\neg(\exists fd. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$

$fd \notin \mathbf{dom}(h.fds)) \wedge$

$(\text{windows_arch } h.arch \implies readfds \neq [] \wedge writefds \neq [] \wedge exceptfds \neq [])$

Description

From thread tid , which is in the RUN state, a $\text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)$ call is made. The time-out is well-formed and no signal mask was set: $sigmask = *$. All of the file descriptors in the sets $readfds$, $writefds$, and $exceptfds$ are greater than the maximum allowed file descriptor in a set for the architecture, FD_SETSIZE , and all of them are valid file descriptors: they are in the host's finite map of file descriptors, $h.fds$.

The call blocks: a $tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)$ transition is made, leaving the thread state $\text{PSELECT2}(readfds, writefds, exceptfds)$.

Variations

| | |
|-------|---|
| WinXP | On WinXP FD_SETSIZE is the maximum number of file descriptors in a set, so none of the sets $readfds$, $writefds$, and $exceptfds$ has more than FD_SETSIZE members. Additionally, all three sets may not be empty. |
|-------|---|

pselect_3 **all: slow nonurgent succeed** Something becomes ready or pselect times out

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{PSELECT2}(readfds, writefds, exceptfds))_d) \rrbracket$$

$$\xrightarrow{\tau} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(readfds'', writefds'', exceptfds''))_{\text{sched_timer}})) \rrbracket$$

$readfds' = \text{filter}(\lambda fd. \exists fid \ ff \ sid \ sock.$
 $fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock = h.socks[sid] \wedge$
 $\text{soreadable } h.arch \ sock) readfds \wedge$
 $writefds' = \text{filter}(\lambda fd. \exists fid \ ff \ sid \ sock.$
 $fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock = h.socks[sid] \wedge$
 $\text{sowriteable } h.arch \ sock) writefds \wedge$
 $exceptfds' = \text{filter}(\lambda fd. \exists fid \ ff \ sid \ sock.$
 $fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock = h.socks[sid] \wedge$
 $\text{soexceptional } h.arch \ sock) exceptfds \wedge$
 $(readfds' \neq [] \vee writefds' \neq [] \vee exceptfds' \neq [] \vee \text{timer_expires } d) \wedge$
 $\text{badreadfds} = \text{filter}(\lambda fd. fd \notin \text{dom}(h.fds)) readfds \wedge$
 $\text{badwritefds} = \text{filter}(\lambda fd. fd \notin \text{dom}(h.fds)) writefds \wedge$
 $\text{badexceptfds} = \text{filter}(\lambda fd. fd \notin \text{dom}(h.fds)) exceptfds \wedge$
if windows_arch $h.arch$ **then**
 $readfds'' = readfds' \wedge writefds'' = writefds' \wedge exceptfds'' = exceptfds'$
else
 $readfds'' = \text{INSERT_ORDERED } readfds' \ readfds \ \text{badreadfds} \wedge$
 $writefds'' = \text{INSERT_ORDERED } writefds' \ writefds \ \text{badwritefds} \wedge$
 $exceptfds'' = \text{INSERT_ORDERED } exceptfds' \ exceptfds \ \text{badexceptfds}$

Description

Thread tid is blocked in state $\text{PSELECT2}(readfds, writefds, exceptfds)$. The call now returns three sets: $readfds''$, $writefds''$, and $exceptfds''$. $readfds''$ is the set of valid file descriptors in $readfds$ that are ready for reading: a blocking $\text{recv}(fd, -, -)$ call would not block; see [soreadable \(p202\)](#) for details. $writefds''$ is the set of valid file descriptors in $writefds$ that are ready for writing: a blocking $\text{send}(fd, -, -)$ call would not block; see [sowriteable \(p202\)](#) for details. $exceptfds''$ is the set of valid file descriptors in $exceptfds$ that have pending exceptional conditions; see [soexceptional \(p203\)](#) for details.

Either one of these three sets is not empty or the timer d , which was set to the timeout value specified when the $\text{pselect}()$ call was made, has expired.

A τ transition is made, leaving the thread state $\text{RET}(\text{OK}(readfds'', writefds'', exceptfds''))$.

Variations

| | |
|---------|--|
| FreeBSD | Invalid file descriptors (ones not in the host's finite map of file descriptors, $h.fds$) may be present in the sets $readfds$, $writefds$, and $exceptfds$, and all such file descriptors will then be included in the return sets $readfds''$, $writefds''$, and $exceptfds''$. |
|---------|--|

pselect_4 **all: fast fail** Fail with EINTR: Timeout not well-formed

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$$

$$\overline{tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}$$

$$h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINTR}))_{\text{sched_timer}}) \rangle$$

$\neg(\text{tlopt_wf } timeout)$

Description

From thread *tid*, which is in the RUN state, a `pselect(readfds, writefds, exceptfds, timeout, sigmask)` call is made. The *timeout* value is not well-formed: $timeout = \uparrow(s, ns)$ where either *s* is negative; *ns* is negative; or $ns > 1000000000$. The call fails with an EINTR error.

A `tid`-`pselect(readfds, writefds, exceptfds, timeout, sigmask)` transition is made, leaving the thread state `RET(FAIL EINTR)`.

Model details

Such negative values are not admitted by the POSIX interface type but are by the model interface type (with `(int * int)` option timeouts), so we check and generate EINTR in the wrapper.

pselect_5 **all: fast fail** Fail with EINTR: File descriptor out of range

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$$

$$\overline{tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}$$

$$h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINTR}))_{\text{sched_timer}}) \rangle$$

$$(\exists fd \ n. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$$

$$\text{if } windows_arch \ h.arch$$

$$\text{then } n = \max(\text{length } readfds)(\max(\text{length } writefds)(\text{length } exceptfds)) \wedge$$

$$n \geq \text{FD_SETSIZE } h.arch$$

$$\text{else}$$

$$fd = \text{FD } n \wedge$$

$$n \geq \text{FD_SETSIZE } h.arch) \vee$$

$$(windows_arch \ h.arch \wedge readfds = [] \wedge writefds = [] \wedge exceptfds = [])$$

Description

From thread *tid*, which is in the RUN state, a `pselect(readfds, writefds, exceptfds, timeout, sigmask)` call is made. One or more of the file descriptors in *readfds*, *writefds*, or *exceptfds* is greater than the architecture dependent `FD_SETSIZE`, the maximum file descriptor that can be specified in a `pselect()` call. The call fails with an EINTR error.

A `tid`-`pselect(readfds, writefds, exceptfds, timeout, sigmask)` transition is made, leaving the thread state `RET(FAIL EINTR)`.

Variations

| | |
|-------|--|
| WinXP | On WinXP <code>FD_SETSIZE</code> is the maximum number of file descriptors in a set, so one of the sets <i>readfds</i> , <i>writefds</i> , or <i>exceptfds</i> has more than <code>FD_SETSIZE</code> members. Also, the call will fail with EINTR if the sets <i>readfds</i> , <i>writefds</i> , and <i>exceptfds</i> are all empty. |
|-------|--|

pselect_6 **all: fast fail** **Fail with EBADF or ENOTSOCK: Bad file descriptor**

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{pselect}(readfds, writefds, exceptfds, timeout, sigmask)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}) \rangle$$

$\neg \text{bsd_arch } h.arch \wedge$
 $(\exists fd. (fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$
 $fd \notin \text{dom}(h.fds)) \wedge$
(if windows_arch h.arch then err = ENOTSOCK
else err = EBADF)

Description

From thread *tid*, which is in the RUN state, a `pselect(readfds, writefds, exceptfds, timeout, sigmask)` call is made. There exists a file descriptor *fd* in *readfds*, *writefds*, or *exceptfds* that is not a valid file descriptor. The call fails with an EBADF error on FreeBSD and Linux and an ENOTSOCK error on WinXP.

A `tid`·`pselect(readfds, writefds, exceptfds, timeout, sigmask)` transition is made, leaving the thread state `RET(FAIL err)` where *err* is one of the above errors.

Variations

| | |
|---------|--|
| FreeBSD | This rule does not apply. |
| Linux | As above: the call fails with an EBADF error. |
| WinXP | As above: the call fails with an ENOTSOCK error. |

15.19 recv() (TCP only)

`recv : fd * int * msgbflag list → (string * ((ip * port) * bool) option)`

A call to `recv(fd, n, opts)` reads data from a socket's receive queue. This section describes the behaviour for TCP sockets. Here *fd* is a file descriptor referring to a TCP socket to read data from, *n* is the number of bytes of data to read, and *opts* is a list of message flags. Possible flags are:

- `MSG_DONTWAIT`: Do not block if there is no data available.
- `MSG_OOB`: Return out-of-band data.
- `MSG_PEEK`: Read data but do not remove it from the socket's receive queue.
- `MSG_WAITALL`: Block until all *n* bytes of data are available.

The returned `string` is the data read from the socket's receive queue. The `((ip * port) * bool) option` is always returned as `*` for a TCP socket.

In order to receive data, a TCP socket must be connected to a peer; otherwise, the `recv()` call will fail with an `ENOTCONN` error. If the socket has a pending error then the `recv()` call will fail with this error even if there is data available.

If there is no data available and non-blocking behaviour is not enabled (the socket's `O_NONBLOCK` flag is not set and the `MSG_DONTWAIT` flag was not used) then the `recv()` call will block until data arrives or an error occurs. If non-blocking behaviour is enabled and there is no data or error then the call will fail with an `EAGAIN` error.

The `MSG_OOB` flag can be set in order to receive out-of-band data; for this, the socket's `SO_OOBINLINE` cannot be set (i.e. out-of-band data must not be being returned inline).

15.19.1 Errors

A call to `recv()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------|---|
| EAGAIN | Non-blocking recv() call made and no data available; or out-of-band data requested and none is available. |
| EINVAL | Out-of-band data requested and SO_OOBINLINE flag set or the out-of-band data has already been read. |
| ENOTCONN | Socket not connected. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| EINTR | The system was interrupted by a caught signal. |
| ENOBUFS | Out of resources. |
| ENOMEM | Out of resources. |

15.19.2 Common cases

A TCP socket is created and then connected to a peer; a `recv()` call is made to receive data from that peer: *socket_1; return_1; connect_1; return_1; recv_1; ...*

15.19.3 API

```
Posix:    ssize_t recv(int socket, void *buffer, size_t length, int flags);
FreeBSD:  ssize_t recv(int s, void *buf, size_t len, int flags);
Linux:    int  recv(int s, void *buf, size_t len, int flags);
WinXP:    int  recv(SOCKET s, char* buf, int len, int flags);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to receive from, corresponding to the `fd` argument of the model `recv()`.
- `buffer` is a pointer to a buffer to place the received data in, which upon return contains the data received on the socket. This corresponds to the `string` return value of the model `recv()`.
- `length` is the amount of data to be read from the socket, corresponding to the `int` argument of the model `recv()`; it should be at most the length of `buffer`.
- `flags` is a disjunction of the message flags that are set for the call, corresponding to the `msgbflag` list argument of the model `recv()`.
- the returned `ssize_t` is either non-negative, in which case it is the amount of data that was received by the socket, or it is `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

There are other functions used to receive data on a socket. `recvfrom()` is similar to `recv()` except it returns the source address of the data; this is used for UDP but is not necessary for TCP as the source address will always be the peer the socket has connected to. `recvmsg()`, another input function, is a more general form of `recv()`.

15.19.4 Model details

If the call blocks then the thread enters state `RECV2(sid, n, opts)` where:

- `sid` : `sid` is the identifier of the socket that the `recv()` call was made on,
- `n` : `num` is the number of bytes to be read, and
- `opts` : `msgbflag` list is the list of message flags.

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, `EFAULT` can be returned if the `buffer` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model `recv()`.
- In Posix, `EIO` may be returned to indicated that an I/O error occurred while reading from or writing to the file system; this is not modelled here.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

The following Linux message flags are not modelled: `MSG_NOSIGNAL`, `MSG_TRUNC`, and `MSG_ERRQUEUE`.

15.19.5 Summary

| | | |
|----------------------|------------------------------------|---|
| <code>recv_1</code> | tcp: fast succeed | Successfully return data from the socket without blocking |
| <code>recv_2</code> | tcp: block | Block, entering state <code>RECV2</code> as not enough data is available |
| <code>recv_3</code> | tcp: slow nonurgent succeed | Blocked call returns from <code>RECV2</code> state |
| <code>recv_4</code> | tcp: fast fail | Fail with <code>EAGAIN</code> : non-blocking call would block waiting for data |
| <code>recv_5</code> | tcp: fast succeed | Successfully read non-inline out-of-band data |
| <code>recv_6</code> | tcp: fast fail | Fail with <code>EAGAIN</code> or <code>EINVAL</code> : <code>recv()</code> called with <code>MSG_OOB</code> set and out-of-band data is not available |
| <code>recv_7</code> | tcp: fast fail | Fail with <code>ENOTCONN</code> : socket not connected |
| <code>recv_8</code> | tcp: fast fail | Fail with pending error |
| <code>recv_8a</code> | tcp: slow urgent fail | Fail with pending error from blocked state |
| <code>recv_9</code> | tcp: fast fail | Fail with <code>ESHUTDOWN</code> : socket shut down for reading on WinXP |

15.19.6 Rules

```

recv_1  tcp: fast succeed  Successfully return data from the socket without blocking
  h ⟨ts := ts ⊕ (tid ↦ (RUN)d)⟩;
  socks := socks ⊕
    [(sid, SOCK(↑ fid, sf, is1, ps1, is2, ps2, es, cantsndmore, cantrcvmore,
      TCP_Sock(st, cb, *, sndq, sndurp, revq, rcvurp, iobc)))]
tid.recv(fd, n0, opts0) → h ⟨ts := ts ⊕ (tid ↦ (RET(OK(implode str, *)))sched_timer)⟩;
  socks := socks ⊕
    [(sid, SOCK(↑ fid, sf, is1, ps1, is2, ps2, es, cantsndmore, cantrcvmore,
      TCP_Sock(st, cb, *, sndq, sndurp, revq'', rcvurp', iobc)))]

((st ∈ {ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSING;
  TIME_WAIT; CLOSE_WAIT; LAST_ACK}) ∧

```

```

  is1 = ↑ i1 ∧ ps1 = ↑ p1 ∧ is2 = ↑ i2 ∧ ps2 = ↑ p2) ∨
(st = CLOSED)) ∧
n = clip_int_to_num n0 ∧
opts = list_to_set opts0 ∧
fd ∈ dom(h.fds) ∧
fid = h.fds[fd] ∧
h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
MSG_OOB ∉ opts ∧

```

(* We return now if we can fill the buffer, or we can reach the low-water mark (usually ignored if MSG_WAITALL is set), or we can reach EOF or the next urgent-message boundary. Pending errors are not checked. *)

```

let have_all_data = (length rcvq ≥ n) in
let have_enough_data = (length rcvq ≥ sf.n(SO_RCVLOWAT)) in
let partial_data_ok = (MSG_WAITALL ∉ opts ∨ n > sf.n(SO_RCVBUF) ∨
  (¬(bsd_arch h.arch) ∧ MSG_PEEK ∈ opts)) in
let urgent_data_ahead = (∃om.rcvurp = ↑ om ∧ 0 < om ∧ om ≤ length rcvq) in
(have_all_data ∨ (have_enough_data ∧ partial_data_ok) ∨ urgent_data_ahead ∨ cantrcvmore) ∧

```

```

((str, rcvq') = SPLIT(min n
  (case rcvurp of
    * → length rcvq ||
    ↑ om → if om = 0 then (length rcvq)
      else min om(length rcvq)))
  rcvq) ∧
rcvq'' = (if MSG_PEEK ∈ opts then rcvq else rcvq') ∧
rcvurp' = (case rcvurp of
  * → * ||
  ↑ om → if om = 0 then *
    else if om ≤ length str then ↑ 0 else ↑(om - length str))

```

Description

From thread *tid*, which is in the RUN state, a `rcv(fd, n0, opts0)` call is made where out-of-band data is not requested. *fd* refers to a synchronised TCP socket *sid* with binding quad (↑ *i*₁, ↑ *p*₁, ↑ *i*₂, ↑ *p*₂) and no pending error. Alternatively the socket is uninitialised and in state CLOSED.

The call can return immediately because either: (1) there are at least *n* bytes of data in the socket's receive queue (the *have_all_data* case above); (2) the length of the socket's receive queue is greater than or equal to the minimum number of bytes for socket `rcv()` operations, *sf.n(SO_RCVLOWAT)*, and the call does not have to return all *n* bytes of data; either because (i) the MSG_WAITALL flag is not set in *opts₀*, (ii) the number of bytes requested is greater than the number of bytes in the socket's receive queue, or (iii) on non-FreeBSD architectures the MSG_PEEK flag is set in *opts₀* (the *have_enough_data* ∧ *partial_data_ok* case above); (3) there is urgent data available in the socket's receive queue (the *urgent_data_ahead* case above); or (4) the socket has been shutdown for reading.

The call succeeds, returning a string, **implode** *str*, which is either: (5) the smaller of the first *n* bytes of the socket's receive queue or its entire receive queue, if the urgent pointer is not set or the socket is at the urgent mark; or (6) the smaller of the first *n* bytes of the the socket's receive queue, the data in its receive queue up to the urgent mark, and its entire receive queue, if the urgent mark is set and the socket is not at the urgent mark.

A *tid.rcv(fd, n₀, opts₀)* transition is made leaving the thread state RET(OK(**implode** *str*, *)). If the MSG_PEEK flag was set in *opts₀* then the socket's receive queue remains unchanged; otherwise, the data *str* is removed from the head of the socket's receive queue, *rcvq*, to leave the socket with new receive queue *rcvq'*. If the receive urgent pointer was not set or was set to ↑ 0 then it will be set to *; if it was set to ↑ *om* and *om* is less than the length of the returned string then it will be set to ↑ 0 (because the returned string was the data in the receive queue up to the urgent mark); otherwise it will be set to ↑(*om* - **length** *str*).

Model details

The amount of data requested, *n₀*, is clipped to a natural number from an integer, using `clip_int_to_num`.

POSIX specifies an unsigned type for n_0 and this is one possible model thereof.

The $opts_0$ argument to `recv()` is of type `msgbflag list`, but it is converted to a set, $opts$, using `list_to_set`.

The data itself is represented as a *byte list* in the datagram but is returned a `string`: the `implode` function is used to do the conversion.

recv_2 **tcp: block** **Block, entering state Recv2 as not enough data is available**

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid \cdot \text{recv}(fd, n_0, opts_0)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RCV2}(sid, n, opts))_{\text{never_timer}}) \rrbracket$$

$n = \text{clip_int_to_num } n_0 \wedge$
 $opts = \text{list_to_set } opts_0 \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \text{cantsndmore}, \text{cantrcvmore},$
 $\text{TCP_Sock}(st, cb, *, \text{sndq}, \text{sndurp}, \text{rcvq}, \text{rcvurp}, \text{iobc})) \wedge$
 $st \in \{\text{ESTABLISHED}; \text{SYN_SENT}; \text{SYN_RECEIVED}; \text{FIN_WAIT_1}; \text{FIN_WAIT_2}\} \wedge$
 $\text{MSG_OOB} \notin opts \wedge$

(* We block if not enough (see *recv_1* (p209)) data is available and there is no pending error. *)

let *blocking* = $\neg(\text{MSG_DONTWAIT} \in opts \vee ff.b(\text{O_NONBLOCK}))$ **in**

let *have_all_data* = $(\text{length } rcvq \geq n)$ **in**

let *have_enough_data* = $(\text{length } rcvq \geq sf.n(\text{SO_RCVLOWAT}))$ **in**

let *partial_data_ok* = $(\text{MSG_WAITALL} \notin opts \vee n > sf.n(\text{SO_RCVBUF}) \vee$
 $(\neg(\text{bsd_arch } h.arch) \wedge \text{MSG_PEEK} \in opts))$ **in**

let *urgent_data_ahead* = $(\exists om. rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \text{length } rcvq)$ **in**

blocking \wedge

$\neg(\text{have_all_data} \vee (\text{have_enough_data} \wedge \text{partial_data_ok}) \vee \text{urgent_data_ahead} \vee \text{cantrcvmore}) \wedge$

es = *

Description

From thread tid , which is in the `RUN` state, a `recv(fd, n0, opts0)` call is made where out-of-band data is not requested. fd refers to a TCP socket sid in state `ESTABLISHED`, `SYN_SENT`, `SYN_RECEIVED`, `FIN_WAIT_1`, or `FIN_WAIT_2`, with binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$ and no pending error. The call is blocking: the `MSG_DONTWAIT` flag is not set in $opts_0$ and the socket's `O_NONBLOCK` flag is not set.

The call cannot return immediately because: (1) there are less than n bytes of data in the socket's receive queue; (2) there are less than $sf.n(\text{SO_RVLOWAT})$ (the minimum number of bytes for socket `recv()` operations) bytes of data in the socket's receive queue or the call must return all n bytes of data: (i) the `MSG_WAITALL` flag is set in $opts_0$, (ii) the number of bytes requested is greater than the length of the socket's receive queue, and (iii) the `MSG_PEEK` flag is not set in $opts_0$; (3) there is no urgent data ahead in the socket's receive queue; and (4) the socket is not shutdown for reading.

The call blocks in state `RCV2` waiting for data; a $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state `RCV2(sid, n, opts)`.

Model details

The amount of data requested, n_0 , is clipped to a natural number from an integer, using `clip_int_to_num`. POSIX specifies an unsigned type for n_0 , whereas the model uses `int`.

The $opts_0$ argument to `recv()` is of type `msgbflag list`, but it is converted to a set, $opts$, using `list_to_set`.

Variations

| | |
|---------|--|
| FreeBSD | In case (iii) above, the <code>MSG_PEEK</code> flag may be set in $opts_0$. |
|---------|--|

recv_3 **tcp: slow nonurgent succeed** Blocked call returns from Recv2 state

$$\begin{aligned}
 & h \llbracket ts := ts \oplus (tid \mapsto (\text{RECV2}(sid, n, opts))_d); \\
 & socks := socks \oplus \\
 & \quad \llbracket (sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \\
 & \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc))) \rrbracket \\
 \xrightarrow{\tau} & h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\mathbf{implode} \ str, *)))_{\text{sched_timer}}); \\
 & socks := socks \oplus \\
 & \quad \llbracket (sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \\
 & \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq'', rcvurp', iobc))) \rrbracket
 \end{aligned}$$

$$\begin{aligned}
 & ((st \in \{\text{ESTABLISHED}; \text{FIN_WAIT_1}; \text{FIN_WAIT_2}; \text{CLOSING}; \\
 & \quad \text{TIME_WAIT}; \text{CLOSE_WAIT}; \text{LAST_ACK}\} \wedge \\
 & \quad is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2) \vee \\
 & st = \text{CLOSED}) \wedge
 \end{aligned}$$

(* We return at last if we now have enough (see *recv_1* (p209)) data available. Pending errors are not checked. *)

```

let have_all_data = (length rcvq ≥ n) in
let have_enough_data = (length rcvq ≥ sf.n(SO_RCVLOWAT)) in
let partial_data_ok = (MSG_WAITALL ∉ opts ∨ n > sf.n(SO_RCVBUF) ∨
  (¬(bsd_arch h.arch) ∧ MSG_PEEK ∈ opts)) in
let urgent_data_ahead = (∃om.rcvurp = ↑ om ∧ 0 < om ∧ om ≤ length rcvq) in
(have_all_data ∨ (have_enough_data ∧ partial_data_ok) ∨ urgent_data_ahead ∨ cantrcvmore) ∧

```

$$\begin{aligned}
 (str, rcvq') &= \text{SPLIT}(\mathbf{min} \ n \\
 & \quad (\mathbf{case} \ rcvurp \ \mathbf{of} \\
 & \quad \quad * \rightarrow \mathbf{length} \ rcvq \ || \\
 & \quad \quad \uparrow om \rightarrow \mathbf{if} \ om = 0 \ \mathbf{then} \ (\mathbf{length} \ rcvq) \\
 & \quad \quad \quad \mathbf{else} \ \mathbf{min} \ om(\mathbf{length} \ rcvq)) \\
 & \quad rcvq \wedge \\
 rcvq'' &= (\mathbf{if} \ \text{MSG_PEEK} \in \ opts \ \mathbf{then} \ rcvq \ \mathbf{else} \ rcvq') \wedge \\
 rcvurp' &= (\mathbf{case} \ rcvurp \ \mathbf{of} \\
 & \quad * \rightarrow * \ || \\
 & \quad \uparrow om \rightarrow \mathbf{if} \ om = 0 \ \mathbf{then} \ * \\
 & \quad \quad \mathbf{else} \ \mathbf{if} \ om \leq \mathbf{length} \ str \ \mathbf{then} \ \uparrow 0 \ \mathbf{else} \ \uparrow(om - \mathbf{length} \ str))
 \end{aligned}$$

Description

Thread *tid* is in the `RECV2(sid, n, opts)` state after a previous `recv()` call blocked. *sid* refers either to a synchronised TCP socket with binding quad ($\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2$); or to a TCP socket in state `CLOSED`.

Sufficient data is not available on the socket for the call to return: either (1) there is at least *n* bytes of data in the socket's receive queue (the *have_all_data* case above); (2) the length of the socket's receive queue is greater than or equal to the minimum number of bytes for socket `recv()` operations, `sf.n(SO_RCVLOWAT)`, and the call does not have to return all *n* bytes of data (the *partial_data_ok* case): either (i) the `MSG_WAITALL` flag is not set in *opts*, (ii) the number of bytes requested is greater than the number of bytes in the socket's receive queue, or (iii) on non-FreeBSD architectures the `MSG_PEEK` flag is set in *opts* (the *have_enough_data* ∧ *partial_data_ok* case above); (3) there is urgent data available in the socket's receive queue (the *urgent_data_ahead* case above); or (4) the socket has been shutdown for reading.

The data returned, *str*, is either: (1) the smaller of the first *n* bytes of the socket's receive queue or its entire receive queue, if the urgent pointer is not set or the socket is at the urgent mark; or (2) the smaller of the first *n* bytes of the the socket's receive queue, the data in its receive queue up to the urgent mark, and its entire receive queue, if the urgent mark is set and the socket is not at the urgent mark.

A τ transition is made leaving the thread state `RET(OK(implode str, *))`. If the `MSG_PEEK` flag was set in *opts* then the socket's receive queue remains unchanged; otherwise, the data *str* is removed from the head of the socket's receive queue, *rcvq*, to leave the socket with new receive queue *rcvq'*. If the receive urgent pointer was not set or was set to $\uparrow 0$ then it will be set to $*$; if it was set to $\uparrow om$ and *om* is less than the

length of the returned string then it will be set to $\uparrow 0$ (because the returned string was the data in the receive queue up to the urgent mark); otherwise it will be set to $\uparrow(om - \mathbf{length} \text{ } str)$.

Model details

The data itself is represented as a *byte list* in the datagram but is returned a *string*: the **implode** function is used to do the conversion.

recv_4 **tcp: fast fail** Fail with EAGAIN: non-blocking call would block waiting for data

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\mathbf{RUN})_d) \rangle}{tid \cdot \mathbf{recv}(fd, n_0, opts_0)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\mathbf{RET}(\mathbf{FAIL} \ \mathbf{EAGAIN}))_{\text{sched_timer}}) \rangle$$

$n = \mathbf{clip_int_to_num} \ n_0 \wedge$
 $opts = \mathbf{list_to_set} \ opts_0 \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge$
 $h.socks[sid] = \mathbf{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{cantsndmore}, \mathbf{cantrcvmore},$
 $\quad \mathbf{TCP_Sock}(st, cb, *, \mathbf{sndq}, \mathbf{sndurp}, \mathbf{rcvq}, \mathbf{rcvurp}, \mathbf{iobc})) \wedge$
 $st \in \{\mathbf{ESTABLISHED}; \mathbf{SYN_SENT}; \mathbf{SYN_RECEIVED}; \mathbf{FIN_WAIT_1}; \mathbf{FIN_WAIT_2}\} \wedge$
 $\mathbf{MSG_OOB} \notin opts \wedge$

(* We fail if we would otherwise block (see *recv_2* (p211); these conditions are identical). *)

let *blocking* = $\neg(\mathbf{MSG_DONTWAIT} \in opts \vee ff.b(\mathbf{O_NONBLOCK}))$ **in**
let *have_all_data* = $(\mathbf{length} \ \mathbf{rcvq} \geq n)$ **in**
let *have_enough_data* = $(\mathbf{length} \ \mathbf{rcvq} \geq sf.n(\mathbf{SO_RCVLOWAT}))$ **in**
let *partial_data_ok* = $(\mathbf{MSG_WAITALL} \notin opts \vee n > sf.n(\mathbf{SO_RCVBUF}) \vee$
 $\quad (\neg(\mathbf{bsd_arch} \ h.arch) \wedge \mathbf{MSG_PEEK} \in opts))$ **in**
let *urgent_data_ahead* = $(\exists om. \mathbf{rcvurp} = \uparrow om \wedge 0 < om \wedge om \leq \mathbf{length} \ \mathbf{rcvq})$ **in**
 $\neg \mathbf{blocking} \wedge$
 $\neg(\mathbf{have_all_data} \vee (\mathbf{have_enough_data} \wedge \mathbf{partial_data_ok}) \vee \mathbf{urgent_data_ahead} \vee \mathbf{cantrcvmore}) \wedge$
 $(\mathbf{rcvq} = [] \implies es = *)$

Description

From thread *tid*, which is in the RUN state, a $\mathbf{recv}(fd, n_0, opts_0)$ call is made where out-of-band data is not requested. *fd* refers to a TCP socket *sid* with binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$ and no pending error, which is in state ESTABLISHED, SYN_SENT, SYN_RECEIVED, FIN_WAIT_1, or FIN_WAIT_2. The $\mathbf{recv}()$ call is non-blocking: either the MSG_DONTWAIT flag was set in *opts_0* or the socket's O_NONBLOCK flag is set.

The call would block because: (1) there are less than *n* bytes of data in the socket's receive queue; (2) there are less than $sf.n(\mathbf{SO_RVLOWAT})$ (the minimum number of bytes for socket $\mathbf{recv}()$ operations) bytes of data in the socket's receive queue or the call must return all *n* bytes of data: (i) the MSG_WAITALL flag is set in *opts_0*, (ii) the number of bytes requested is greater than the length of the socket's receive queue, and (iii) the MSG_PEEK flag is not set in *opts_0*; (3) there is no urgent data ahead in the socket's receive queue; (4) the socket is not shutdown for reading; and (5) if the socket's receive queue is empty then it has no pending error.

The call fails with an EAGAIN error. A $tid \cdot \mathbf{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\mathbf{RET}(\mathbf{FAIL} \ \mathbf{EAGAIN})$.

Model details

The amount of data requested, *n_0*, is clipped to a natural number from an integer, using $\mathbf{clip_int_to_num}$. POSIX specifies an unsigned type for *n_0* and this is one possible model thereof.

The *opts_0* argument to $\mathbf{recv}()$ is of type *msgbflag list*, but it is converted to a set, *opts*, using **list_to_set**.

Variations

| | |
|---------|---|
| FreeBSD | In case (iii) above, the MSG_PEEK flag may be set in $opts_0$. |
|---------|---|

recv_5 **tcp: fast succeed** Successfully read non-inline out-of-band data

$$\begin{aligned}
& h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket; \\
& socks := socks \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantndmore, cantrcvmore, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))] \\
\hline
& \text{tid} \cdot \text{recv}(fd, n_0, opts_0) \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } str, *)))_{\text{sched_timer}}) \rrbracket; \\
& \quad socks := socks \oplus \\
& \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantndmore, cantrcvmore, \\
& \quad \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc')))]
\end{aligned}$$

$$\begin{aligned}
n &= \text{clip_int_to_num } n_0 \wedge \\
opts &= \text{list_to_set } opts_0 \wedge \\
fd &\in \text{dom}(h.fds) \wedge \\
fid &= h.fds[fd] \wedge \\
h.files[fd] &= \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
\text{MSG_OOB} &\in opts \wedge \\
\neg sf.b(\text{SO_OOBINLINE}) &\wedge \\
iobc &= \text{OOBDATA } c \wedge \\
str &= (\text{if } n = 0 \text{ then } [] \text{ else } [c]) \wedge \\
iobc' &= (\text{if } \text{MSG_PEEK} \in opts \text{ then } iobc \text{ else } \text{HAD_OOBDATA})
\end{aligned}$$

Description

From thread tid , which is in the RUN state, a $\text{recv}(fd, n_0, opts_0)$ call is made. fd refers to a TCP socket sid with binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$ and no pending error. Out-of-band data is requested: the MSG_OOB flag is set in $opts_0$, and out-of-band data is not being returned inline: $\neg sf.b(\text{SO_OOBINLINE})$. There is a byte c of out-of-band data on the socket; if zero bytes of data were requested, $n_0 = 0$, then the empty string is returned, otherwise c is returned.

A $\text{tid} \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{implode } str, *))$ where **implode** str is the returned out-of-band data. If the MSG_PEEK flag was set in $opts_0$ then the byte of out-of-band data is left in place, $iobc' = iobc$; otherwise it is removed and marked as read: $iobc' = \text{HAD_OOBDATA}$.

Model details

The amount of data requested, n_0 , is clipped to a natural number from an integer, using clip_int_to_num . POSIX specifies an unsigned type for n_0 , whereas the model uses int .

The $opts_0$ argument to $\text{recv}()$ is of type msgbflag list , but it is converted to a set, $opts$, using list_to_set .

The data itself is represented as a *byte list* in the datagram but is returned a *string*: the **implode** function is used to do the conversion.

recv_6 **tcp: fast fail** Fail with EAGAIN or EINVAL: $\text{recv}()$ called with MSG_OOB set and out-of-band data is not available

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{\text{tid} \cdot \text{recv}(fd, n_0, opts_0)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rrbracket$$

$$\begin{aligned}
n &= \text{clip_int_to_num } n_0 \wedge \\
opts &= \text{list_to_set } opts_0 \wedge \\
fd &\in \text{dom}(h.fds) \wedge \\
fid &= h.fds[fd] \wedge \\
h.files[fd] &= \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge
\end{aligned}$$

```

h.socks[sid] = SOCK( $\uparrow$  fd, sf,  $\uparrow$  i1,  $\uparrow$  p1,  $\uparrow$  i2,  $\uparrow$  p2, *, cantsndmore, cantrcvmore,
TCP_Sock(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc))  $\wedge$ 
MSG_OOB  $\in$  opts  $\wedge$ 
(if sf.b(SO_OOBINLINE)
then (e = EINVAL)
else case iobc of

    NO_OOBDATA  $\rightarrow$  (e = if rcvurp = * then EINVAL else EAGAIN) ||
    OOBDATA c  $\rightarrow$  F ||
    HAD_OOBDATA  $\rightarrow$  (e = EINVAL))

```

Description

From thread *tid*, which is in the RUN state, a `recv(fd, n0, opts0)` call is made. *fd* refers to a TCP socket identified by *sid* with binding quad (\uparrow *i*₁, \uparrow *p*₁, \uparrow *i*₂, \uparrow *p*₂) and no pending error. The MSG_OOB flag is set in *opts*₀, indicating that out-of-band data should be returned, but no out-of-band data is available because either: (1) out-of-band data is being returned in-line (the *sf*.*b*(SO_OOBINLINE) flag is set); (2) the out-of-band data on the socket has already been read; (3) there is no out-of-band data and the receive urgent pointer is set; or (4) there is no out-of-band data but the urgent pointer is set, corresponding to the case where the peer has advertised urgent data but that data has yet to arrive. The call fails with an EINVAL error in cases (1), (2), and (3); and a EAGAIN error in case (4) indicating that the `recv()` call should be made again to see if the data has now arrived.

A *tid*.`recv(fd, n0, opts0)` transition is made, leaving the thread state RET(FAIL *e*) where *e* is one of the above errors.

recv_7 **tcp: fast fail** Fail with ENOTCONN: socket not connected

```

h ( $\llbracket$  ts := ts  $\oplus$  (tid  $\mapsto$  (RUN)d)  $\rrbracket$ )
 $\xrightarrow{tid \cdot \text{recv}(fd, n_0, opts_0)}$  h ( $\llbracket$  ts := ts  $\oplus$  (tid  $\mapsto$  (RET(FAIL ENOTCONN))sched_timer)  $\rrbracket$ )

```

```

fd  $\in$  dom(h.fds)  $\wedge$ 
fid = h.fds[fd]  $\wedge$ 
h.files[fid] = FILE(FT_SOCKET(sid), ff)  $\wedge$ 
sock = h.socks[sid]  $\wedge$ 
TCP_PROTO(tcp_sock) = sock.pr  $\wedge$ 
(tcp_sock.st = LISTEN  $\vee$ 
tcp_sock.st = CLOSED  $\wedge$  sock.cantrcvmore = F)

```

)

Description

From thread *tid*, which is in the RUN state, a `recv(fd, n0, opts0)` call is made. *fd* refers to a TCP socket *sock* identified by *sid* which is either in the LISTEN state or is not shutdown for reading in the CLOSED state. The call fails with an ENOTCONN error.

A *tid*.`recv(fd, n0, opts0)` transition is made, leaving the thread state RET(FAIL ENOTCONN).

recv_8 **tcp: fast fail** Fail with pending error

```

h ( $\llbracket$  ts := ts  $\oplus$  (tid  $\mapsto$  (RUN)d) ;
socks := socks  $\oplus$ 
 $\llbracket$  (sid, SOCK( $\uparrow$  fd, sf, is1, ps1, is2, ps2,  $\uparrow$  e, cantsndmore, cantrcvmore, TCP_PROTO(tcp_sock)))  $\rrbracket$   $\rrbracket$ )
 $\xrightarrow{tid \cdot \text{recv}(fd, n_0, opts_0)}$ 

```



```

h {ts := ts  $\oplus$  (tid  $\mapsto$  (RET(FAIL e))sched_timer);
socks := socks  $\oplus$ 
  [(sid, SOCK( $\uparrow$  fid, sf, is1, ps1, is2, ps2, es, cantndmore, cantrcvmore, TCP_PROTO(tcp_sock)))]

```

```

opts = list_to_set opts0  $\wedge$ 
n = clip_int_to_num n0  $\wedge$ 
fd  $\in$  dom(h.fds)  $\wedge$ 
fid = h.fds[fd]  $\wedge$ 
h.files[fd] = FILE(FT_SOCKET(sid), ff)  $\wedge$ 
((tcp_sock.st  $\notin$  {CLOSED; LISTEN}  $\wedge$  is2 =  $\uparrow$  i2  $\wedge$  ps2 =  $\uparrow$  p2)  $\vee$ 
tcp_sock.st = CLOSED)  $\wedge$ 

```

(* We fail immediately if there is a pending error and we could not otherwise return data (see *recv_1* (p209)). *)

```

let rcvq = tcp_sock.rcvq in
let rcvurp = tcp_sock.rcvurp in
let blocking =  $\neg$ (MSG_DONTWAIT  $\in$  opts  $\vee$  ff.b(O_NONBLOCK)) in
let have_all_data = (length rcvq  $\geq$  n) in
let have_enough_data = (length rcvq  $\geq$  sf.n(SO_RCVLOWAT)) in
let partial_data_ok = (MSG_WAITALL  $\notin$  opts  $\vee$  n > sf.n(SO_RCVBUF)  $\vee$ 
  ( $\neg$ (bsd_arch h.arch)  $\wedge$  MSG_PEEK  $\in$  opts)) in
let urgent_data_ahead = ( $\exists$ om.rcvurp =  $\uparrow$  om  $\wedge$  0 < om  $\wedge$  om  $\leq$  length rcvq) in
 $\neg$ (have_all_data  $\vee$  (have_enough_data  $\wedge$  partial_data_ok)  $\vee$  urgent_data_ahead)  $\wedge$ 
(blocking  $\vee$  rcvq = [])  $\wedge$ 

```

```

es = if MSG_PEEK  $\in$  opts then  $\uparrow$  e else *

```

Description

From thread *tid*, which is in the RUN state, a *recv*(*fd*, *n*₀, *opts*₀) call is made. *fd* refers to a TCP socket that either is in state CLOSED or is in state other than CLOSED or LISTEN with peer address set to (\uparrow *i*₂, \uparrow *p*₂). The socket has a pending error *e*.

The call cannot immediately return data because: (1) there are less than *n* bytes of data in the socket's receive queue; (2) there are less than *sf.n(SO_RVLOWAT)* (the minimum number of bytes for socket *recv*() operations) bytes of data in the socket's receive queue or the call must return all *n* bytes of data: (i) the MSG_WAITALL flag is set in *opts*₀, (ii) the number of bytes requested is greater than the length of the socket's receive queue, and (iii) the MSG_PEEK flag is not set in *opts*₀; (3) there is no urgent data ahead in the socket's receive queue; and (4) either the call is a blocking one: the MSG_DONTWAIT flag is set in *opts*₀ or the socket's O_NONBLOCK flag is set, or the socket's receive queue is empty.

The call fails, returning the pending error. A *tid.recv*(*fd*, *n*₀, *opts*₀) transition is made, leaving the thread state RET(FAIL *e*). If the MSG_PEEK flag was set in *opts*₀ then the socket's pending error remains, otherwise it is cleared.

Model details

The *opts*₀ argument to *recv*() is of type *msgbflag list*, but it is converted to a set, *opts*, using *list_to_set*.

Variations

| | |
|---------|---|
| FreeBSD | In case (iii) above, the MSG_PEEK flag may be set in <i>opts</i> ₀ . |
|---------|---|

recv_8a **tcp: slow urgent fail** Fail with pending error from blocked state

```

h {ts := ts  $\oplus$  (tid  $\mapsto$  (RECV2(sid, n, opts))d);
socks := socks  $\oplus$ 
  [(sid, sock {es :=  $\uparrow$  e; pr := TCP_PROTO(tcp_sock)})]

```

$$\begin{aligned} \xrightarrow{\tau} & h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}); \\ & \text{socks} := \text{socks} \oplus \\ & \llbracket (sid, \text{sock} \llbracket es := es; pr := \text{TCP_PROTO}(tcp_sock) \rrbracket) \rrbracket \end{aligned}$$

(* We fail now if there is a pending error and we could not otherwise return data (see *recv_1* (p209)). *)

let *have_all_data* = (**length** *tcp_sock.recvq* ≥ *n*) **in**
let *have_enough_data* = (**length** *tcp_sock.recvq* ≥ *sock.sf.n*(SO_RCVLOWAT)) **in**
let *partial_data_ok* = (MSG_WAITALL ∉ *opts* ∨ *n* > *sock.sf.n*(SO_RCVBUF) ∨
 (¬(bsd_arch *h.arch*) ∧ MSG_PEEK ∈ *opts*)) **in**
let *urgent_data_ahead* = (∃*om.tcp_sock.recvurp* = ↑ *om* ∧ 0 < *om* ∧ *om* ≤ **length** *tcp_sock.recvq*) **in**
 ¬(*have_all_data* ∨ (*have_enough_data* ∧ *partial_data_ok*) ∨ *urgent_data_ahead*) ∧

(*es* = **if** MSG_PEEK ∈ *opts* **then** ↑ *e* **else** *)

Description

Thread *tid* is blocked in state *RCV2(sid, n, opts)* where *sid* identifies a socket with pending error ↑ *e*. The call fails, returning the pending error. Data cannot be returned because: (1) there are less than *n* bytes of data in the socket's receive queue; (2) there are less than *sf.n*(SO_RVLOWAT) (the minimum number of bytes for socket *recv()* operations) bytes of data in the socket's receive queue or the call must return all *n* bytes of data: (i) the MSG_WAITALL flag is set in *opts*, (ii) the number of bytes requested is greater than the length of the socket's receive queue, and (iii) the MSG_PEEK flag is not set in *opts*; and (3) there is no urgent data ahead in the socket's receive queue.

The thread returns from the blocked state, returning the pending error. A τ transition is made, leaving the thread state *RET(FAIL e)*. If the MSG_PEEK flag was set in *opts* then the socket's pending error remains, otherwise it is cleared.

Variations

| | |
|---------|--|
| FreeBSD | In case (iii) above, the MSG_PEEK flag may be set in <i>opts</i> . |
|---------|--|

recv_9 **tcp: fast fail** Fail with ESHUTDOWN: socket shut down for reading on WinXP

$$\begin{aligned} & h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ & \text{socks} := \text{socks} \oplus \\ & \llbracket (sid, \text{sock} \llbracket \text{cantrcvmore} := \mathbf{T}; pr := \text{TCP_PROTO}(tcp_sock) \rrbracket) \rrbracket \\ \xrightarrow{tid \cdot \text{recv}(fd, n, opts)} & h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ESHUTDOWN}))_{\text{sched_timer}}); \\ & \text{socks} := \text{socks} \oplus \\ & \llbracket (sid, \text{sock} \llbracket \text{cantrcvmore} := \mathbf{T}; pr := \text{TCP_PROTO}(tcp_sock) \rrbracket) \rrbracket \end{aligned}$$

windows_arch *h.arch* ∧
fd ∈ **dom**(*h.fds*) ∧
fid = *h.fds*[*fd*] ∧
h.files[*fid*] = FILE(FT_SOCKET(*sid*), *ff*)

Description

On WinXP, from thread *tid*, which is in the RUN state, a *recv(fd, n, opts)* call is made where *fd* refers to a TCP socket *sid* which is shut down for reading. The call fails with an ESHUTDOWN error.

A *tid.recv(fd, n₀, opts₀)* transition is made, leaving the thread state *RET(FAIL ESHUTDOWN)*.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.20 recv() (UDP only)

recv : (fd * int * msgbflag list) → (string * ((ip * port) * bool) option)

A call to `recv(fd, n, opts)` returns data from the datagram on the head of a socket's receive queue. This section describes the behaviour for UDP sockets. Here the `fd` argument is a file descriptor referring to the socket to receive data from, `n` specifies the number of bytes of data to read from that socket, and the `opts` argument is a list of flags for the `recv()` call. The possible flags are:

- `MSG_DONTWAIT`: non-blocking behaviour is requested for this call. This flag only has effect on Linux. FreeBSD and WinXP ignore it. See rules `recv_12` and `recv_13`.
- `MSG_PEEK`: return data from the datagram on the head of the receive queue, without removing that datagram from the receive queue.
- `MSG_WAITALL`: do not return until all `n` bytes of data have been read. Linux and FreeBSD ignore this flag. WinXP fails with `EOPNOTSUPP` as this is not meaningful for UDP sockets: the returned data is from only one datagram.
- `MSG_OOB`: return out-of-band data. This flag is ignored on Linux. On WinXP and FreeBSD the call fails with `EOPNOTSUPP` as out-of-band data is not meaningful for UDP sockets.

The returned value of the `recv()` call, (string * ((ip * port) * bool) option), consists of the data read from the socket (the string), the source address of the data (the ip * port), and a flag specifying whether or not all of the datagram's data was read (the bool). The latter two components are wrapped in an option type (for type compatibility with the TCP `recv()`) but are always returned for UDP. The flag only has meaning on WinXP and should be ignored on FreeBSD and Linux.

For a socket to receive data, it must be bound to a local port. On Linux and FreeBSD, if the socket is not bound to a local port, then it is autobound to an ephemeral port when the `recv()` call is made. On WinXP, calling `recv()` on a socket that is not bound to a local port is an `EINVAL` error.

If a non-blocking `recv()` call is made (the socket's `O_NONBLOCK` flag is set) and there are no datagrams on the socket's receive queue, then the call will fail with `EAGAIN`. If the call is a blocking one and the socket's receive queue is empty then the call will block, returning when a datagram arrives or an error occurs.

If the socket has a pending error then on FreeBSD and Linux, the call will fail with that error. On WinXP, errors from ICMP messages are placed on the socket's receive queue, and so the error will only be returned when that message is at the head of the receive queue.

15.20.1 Errors

A call to `recv()` can fail with the errors below, in which case the corresponding exception is raised.

| | |
|------------|---|
| EAGAIN | The call would block and non-blocking behaviour is requested. This is done either via the <code>MSG_DONTWAIT</code> flag being set in the <code>recv()</code> flags or the socket's <code>O_NONBLOCK</code> flag being set. |
| EMSGSIZE | The amount of data requested in the <code>recv()</code> call on WinXP is less than the amount of data in the datagram on the head of the receive queue. |
| EOPNOTSUPP | Operation not supported: out-of-band data is requested on FreeBSD and WinXP, or the <code>MSG_WAITALL</code> flag is set on a <code>recv()</code> call on WinXP. |

| | |
|-----------|--|
| ESHUTDOWN | On WinXP, a <code>recv()</code> call is made on a socket that has been shutdown for reading. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| EINTR | The system was interrupted by a caught signal. |
| ENOBUFS | Out of resources. |
| ENOMEM | Out of resources. |

15.20.2 Common cases

A UDP socket is created and bound to a local address. Other calls are made and datagrams are delivered to the socket; `recv()` is called to read from a datagram: `socket_1; return_1; bind_1; ... recv_11; return_1;`

A UDP socket is created and bound to a local address. `recv()` is called and blocks; a datagram arrives addressed to the socket's local address and is placed on its receive queue; the call returns: `socket_1; return_1; bind_1; ... recv_12; deliver_in_99; deliver_in_udp_1; recv_15; return_1;`

15.20.3 API

```
Posix:    ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
                  int flags, struct sockaddr *restrict address,
                  socklen_t *restrict address_len);
FreeBSD:  ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                  struct sockaddr *from, socklen_t *fromlen);
Linux:    int recvfrom(int s, void *buf, size_t len, int flags,
                  struct sockaddr *from, socklen_t *fromlen);
WinXP:    int recvfrom(SOCKET s, char* buf, int len, int flags,
                  struct sockaddr* from, int* fromlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to receive from, corresponding to the `fd` argument of the model `recv()`.
- `buffer` is a pointer to a buffer to place the received data in, which upon return contains the data received on the socket. This corresponds to the `string` return value of the model `recv()`.
- `length` is the amount of data to be read from the socket, corresponding to the `int` argument of the model `recv()`; it should be at most the length of `buffer`.
- `flags` is a disjunction of the message flags that are set for the call, corresponding to the `msgflag` list argument of the model `recv()`.
- `address` is a pointer to a `sockaddr` structure of length `address_len`, which upon return contains the source address of the data received by the socket corresponding to the `(ip * port)` in the return value of the model `recv()`. For the `AF_INET` sockets used in the model, it is actually a `sockaddr_in` that is used: the `in_addr.s_addr` field corresponds to the `ip` and the `sin_port` field corresponds to the `port`.
- the returned `ssize_t` is either non-negative, in which case it is the the amount of data that was received by the socket, or it is `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

On WinXP, if the data from a datagram is not all read then the call fails with `EMSGSIZE`, but still fills the `buffer` with data. This is modelled by the `bool` flag in the model `recv()`: if it is set to `T` then the call

succeeded and read all of the datagrams's data; if it is set to **F** then the call failed with EMSGSIZE but still returned data.

There are other functions used to receive data on a socket. `recv()` is similar to `recvfrom()` except it does not have the `address` and `address_len` arguments. It is used when the source address of the data does not need to be returned from the call. `recvmsg()`, another input function, is a more general form of `recvfrom()`.

15.20.4 Model details

If the call blocks then the thread enters state `RECV2(sid, n, opts)` where:

- `sid` : `sid` is the identifier of the socket that the `recv()` call was made on,
- `n` : `num` is the number of bytes to be read, and
- `opts` : `msgbflag` list is the set of message flags.

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, `EFAULT` can be returned if the `buffer` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model `recv()`.
- In Posix, `EIO` may be returned to indicated that an I/O error occurred while reading from or writing to the file system; this is not modelled here.
- `EINVAL` may be returned if the `MSG_OOB` flag is set and no out-of-band data is available; out-of-band data does not exist for UDP so this does not apply.
- `ENOTCONN` may be returned if the socket is not connected; this does not apply for UDP as the socket need not have a peer specified to receive datagrams.
- `ETIMEDOUT` can be returned due to a transmission timeout on a connection; UDP is not connection-oriented so this does not apply.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

The following Linux message flags are not modelled: `MSG_NOSIGNAL`, `MSG_TRUNC`, and `MSG_ERRQUEUE`.

15.20.5 Summary

| | | |
|----------------|---------------------------------|---|
| <i>recv_11</i> | udp: fast succeed | Receive data successfully without blocking |
| <i>recv_12</i> | udp: block | Block, entering <code>RECV2</code> state as no datagrams available on socket |
| <i>recv_13</i> | udp: fast fail | Fail with <code>EAGAIN</code> : call would block and socket is non-blocking or, on Linux, non-blocking behaviour has been requested with the <code>MSG_DONTWAIT</code> flag |
| <i>recv_14</i> | udp: fast fail | Fail with <code>EAGAIN</code> , <code>EADDRNOTAVAIL</code> , or <code>ENOBUFS</code> : there are no ephemeral ports left |
| <i>recv_15</i> | udp: slow urgent succeed | Blocked call returns from <code>RECV2</code> state with data |
| <i>recv_16</i> | udp: fast fail | Fail with <code>EOPNOTSUPP</code> : <code>MSG_WAITALL</code> flag not supported on WinXP, or <code>MSG_OOB</code> flag not supported on FreeBSD and WinXP |
| <i>recv_17</i> | udp: rc | Socket shutdown for reading: fail with <code>ESHUTDOWN</code> on WinXP or succeed on Linux and FreeBSD |
| <i>recv_20</i> | udp: rc | Successful partial read of datagram on head of socket's receive queue on WinXP |
| <i>recv_21</i> | udp: fast succeed | Read zero bytes of data from an empty receive queue on FreeBSD |

| | | |
|----------------|-----------------------|--|
| <i>recv_22</i> | udp: fast fail | Fail with EINVAL on WinXP: socket is unbound |
| <i>recv_23</i> | udp: rc | Read ICMP error from receive queue and fail with that error on WinXP |
| <i>recv_24</i> | udp: fast fail | Fail with pending error |

15.20.6 Rules

recv_11 **udp: fast succeed** Receive data successfully without blocking

$$\begin{aligned}
 & h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle; \\
 & socks := socks \oplus \\
 & \quad [(sid, sock \langle pr := \text{UDP_Sock}(recv) \rangle)] \\
 \hline
 & \xrightarrow{tid \cdot \text{recv}(fd, n_0, opts_0)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } data', \uparrow((i_3, ps_3), b))))_{\text{sched_timer}}) \rangle; \\
 & \quad socks := socks \oplus \\
 & \quad \quad [(sid, sock)]
 \end{aligned}$$

$$\begin{aligned}
 & fd \in \text{dom}(h.fds) \wedge \\
 & fid = h.fds[fd] \wedge \\
 & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 & sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_Sock}(recv')) \wedge \\
 & (\neg(\text{linux_arch } h.arch) \implies \text{cantrcvmore} = \mathbf{F}) \wedge \\
 & recv = (\text{DGRAM_MSG}(\langle is := i_3; ps := ps_3; data := data \rangle)) :: recv'' \wedge \\
 & n = \text{clip_int_to_num } n_0 \wedge \\
 & ((\text{length } data \leq n \wedge data = data') \vee \\
 & \quad (\text{length } data > n \wedge data' = \text{TAKE } n \text{ } data \wedge \text{length } data' = n \wedge \neg(\text{windows_arch } h.arch))) \wedge \\
 & (\text{windows_arch } h.arch \implies b = \mathbf{T}) \wedge \\
 & opts = \text{list_to_set } opts_0 \wedge \\
 & recv' = (\text{if } \text{MSG_PEEK} \in opts \text{ then } recv \text{ else } recv'')
 \end{aligned}$$

Description

Consider a UDP socket *sid*, referenced by *fd*. It is not shutdown for reading, has no pending errors, and is bound to local port *p*₁. Thread *tid* is in the RUN state.

The socket's receive queue has a datagram at its head with data *data* and source address *i*₃, *ps*₃. A call *recv*(*fd*, *n*₀, *opts*₀), from thread *tid*, succeeds.

A *tid*·*recv*(*fd*, *n*₀, *opts*₀) transition is made. The thread is left in state *RET*(*OK*(*implode* *data'*, *↑*(*i*₃, *ps*₃))), where *data'* is either:

- all of the data in the datagram, *data*, if the amount of data requested *n*₀ is greater than or equal to the amount of data in the datagram, or
- the first *n*₀ bytes of *data* if *n*₀ is less than the amount of data in the datagram, unless the architecture is WinXP (see below).

If the *MSG_PEEK* option is set in *opts*₀ then the entire datagram stays on the receive queue; the next call to *recv*() will be able to access this datagram. Otherwise, the entire datagram is discarded from the receive queue, even if all of its data has not been read.

Model details

The amount of data requested, *n*₀, is clipped to a natural number from an integer, using *clip_int_to_num*. POSIX specifies an unsigned type for *n*₀ and this is one possible model thereof.

The *opts*₀ argument to *recv*() is of type *msgbflag* list, but it is converted to a set, *opts*, using *list_to_set*.

The data itself is represented as a *byte* list in the datagram but is returned a *string*: the *implode* function is used to do the conversion.

Variations

| | |
|-------|--|
| WinXP | The amount of data in bytes requested, n_0 , must be greater than or equal to the number of bytes of data in the datagram on the head of the receive queue. The boolean b equals T , indicating that all of the datagram's data has been read. Otherwise refer to rule <i>recv_20</i> . |
|-------|--|

recv_12 **udp: block** Block, entering Recv2 state as no datagrams available on socket

$$h_0 \xrightarrow{tid \cdot \text{recv}(fd, n_0, opts_0)} h_0 \langle \langle ts := ts \oplus (tid \mapsto (\text{RCV2}(sid, n, opts))_{\text{never_timer}}); \rangle \rangle;$$

$$socks := h_0.socks \oplus [(sid, sock \langle \langle ps_1 := \uparrow p'_1 \rangle \rangle)];$$

$$bound := bound \rangle$$

$$h_0 = h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \rangle \rangle;$$

$$socks := socks \oplus [(sid, sock)]; \wedge$$

$$fd \in \text{dom}(h_0.fds) \wedge$$

$$fid = h_0.fds[fd] \wedge$$

$$h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$sock = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, *, \text{cantsndmore}, \mathbf{F}, \text{UDP_Sock}([])) \wedge$$

$$p'_1 \in \text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks) \wedge$$

$$(\text{if } sock.ps_1 = * \text{ then } bound = sid :: h_0.bound \text{ else } bound = h_0.bound) \wedge$$

$$\neg((\text{MSG_DONTWAIT} \in opts \wedge \text{linux_arch } h.arch) \vee ff.b(\text{O_NONBLOCK})) \wedge$$

$$(\text{bsd_arch } h.arch \implies \neg(n = 0)) \wedge$$

$$n = \text{clip_int_to_num } n_0 \wedge$$

$$opts = \text{list_to_set } opts_0$$

Description

Consider a UDP socket sid , referenced by fd , that has no pending errors, is not shutdown for reading, has an empty receive queue, and does not have its `O_NONBLOCK` flag set. The socket is either bound to a local port $\uparrow p'_1$ or can be autobound to a local port $\uparrow p'_1$. From thread tid , which in the `RUN` state, a $\text{recv}(fd, n_0, opts_0)$ call is made. Because there are no datagrams on the socket's receive queue, the call will block.

A $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition will be made, leaving the thread state $\text{RCV2}(sid, n, opts)$. If autobinding occurred then sid will be placed on the head of the host's list of bound sockets: $bound = sid :: h_0.bound$.

Model details

The amount of data requested, n_0 , is clipped to a natural number n from an integer, using `clip_int_to_num`. POSIX specifies an unsigned type for n_0 and this is one possible model thereof.

The $opts_0$ argument to $\text{recv}()$ is of type `msgbflag list`, but it is converted to a set, $opts$, using `list_to_set`.

Variations

| | |
|---------|---|
| FreeBSD | As above, with the added condition that the number of bytes requested to be read is not zero. |
| Linux | As above, with the added condition that the <code>MSG_DONTWAIT</code> flag is not set in $opts_0$. |

recv_13 **udp: fast fail** Fail with `EAGAIN`: call would block and socket is non-blocking or, on

Linux, non-blocking behaviour has been requested with the MSG_DONTWAIT flag

$$h_0 \xrightarrow{tid\cdot\text{recv}(fd, n, \text{opts}_0)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched_timer}}); \\ \text{socks} := \text{socks} \oplus \\ [(sid, s \langle es := *; pr := \text{UDP_Sock}([\]\rangle)] \rangle$$

$$h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ \text{socks} := \text{socks} \oplus \\ [(sid, s \langle es := *; pr := \text{UDP_Sock}([\]\rangle)] \rangle \wedge \\ fd \in \mathbf{dom}(h_0.fds) \wedge \\ fid = h_0.fds[fd] \wedge \\ h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ \text{opts} = \mathbf{list_to_set} \text{ opts}_0 \wedge \\ ((\text{MSG_DONTWAIT} \in \text{opts} \wedge \text{linux_arch } h.arch) \vee ff.b(\text{O_NONBLOCK}))$$

Description

Consider a UDP socket sid referenced by fd . It has no pending errors, and an empty receive queue. The socket is non-blocking: its `O_NONBLOCK` flag has been set. From thread tid , in the `RUN` state, a $\text{recv}(fd, n, \text{opts}_0)$ call is made. The call would block because the socket has an empty receive queue, so the call fails with an `EAGAIN` error.

A $tid\cdot\text{recv}(fd, n, \text{opts}_0)$ transition is made, leaving the thread state `RET(FAIL EAGAIN)`.

Model details

The opts_0 argument is of type `list`. In the model it is converted to a `set` opts using `list_to_set`.

Variations

| | |
|-------|---|
| Linux | As above, but the rule also applies if the socket's <code>O_NONBLOCK</code> flag is not set but the <code>MSG_DONTWAIT</code> flag is set in opts_0 . Also, note that <code>EWOULDBLOCK</code> and <code>EAGAIN</code> are aliased on Linux. |
|-------|---|

recv_14 udp: fast fail Fail with EAGAIN, EADDRNOTAVAIL, or ENOBUFS: there are no ephemeral ports left

$$h_0 \xrightarrow{tid\cdot\text{recv}(fd, n, \text{opts})} h_0 \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rangle$$

$$h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ \text{socks} := \text{socks} \oplus \\ [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, *, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_Sock}([\]\rangle))] \rangle \wedge \\ \text{autobind}(*, \text{PROTO_UDP}, h_0.\text{socks}) = \emptyset \wedge \\ e \in \{\text{EAGAIN}; \text{EADDRNOTAVAIL}; \text{ENOBUFS}\} \wedge \\ fd \in \mathbf{dom}(h_0.fds) \wedge \\ fid = h_0.fds[fd] \wedge \\ h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$$

Description

Consider a UDP socket sid , referenced by fd . The socket has no pending errors, an empty receive queue, and binding quad `*, *, *, *`. From thread tid , which is in the `RUN` state, a $\text{recv}(fd, n, \text{opts})$ call is made. There is no ephemeral port to autobind the socket to, so the call fails with either `EAGAIN`, `EADDRNOTAVAIL` or `ENOBUFS`.

A $tid\cdot\text{recv}(fd, n, \text{opts})$ transition is made, leaving the thread state `RET(FAIL e)` where e is one of the above errors.

recv_15 udp: slow urgent succeed Blocked call returns from Recv2 state with data

$$\begin{aligned}
& h \langle ts := ts \oplus (tid \mapsto (\text{RCV2}(sid, n, opts))_d); \\
& socks := socks \oplus \\
& \quad [(sid, sock \langle ps_1 := \uparrow p_1; es := *; pr := \text{UDP_Sock}(rcvq) \rangle)] \rangle \\
\tau \rightarrow & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\mathbf{implode} \ data', \uparrow((i_3, ps_3), b)))_{\text{sched_timer}}); \\
& socks := socks \oplus \\
& \quad [(sid, sock \langle ps_1 := \uparrow p_1; es := *; pr := \text{UDP_Sock}(rcvq') \rangle)] \rangle
\end{aligned}$$

$$\begin{aligned}
rcvq &= (\text{DGRAM_MSG}(\langle is := i_3; ps := ps_3; data := data \rangle)) :: rcvq'' \wedge \\
(rcvq' &= \mathbf{if} \ \text{MSG_PEEK} \in \text{opts} \ \mathbf{then} \ rcvq \ \mathbf{else} \ rcvq'') \wedge \\
& ((\mathbf{length} \ data \leq n \wedge data = data') \vee \\
& (\mathbf{length} \ data > n \wedge \neg(\text{windows_arch} \ h.\text{arch}) \wedge data' = \text{TAKE} \ n \ data' \wedge \mathbf{length} \ data' = n)) \wedge \\
& (\text{windows_arch} \ h.\text{arch} \implies b = \mathbf{T})
\end{aligned}$$

Description

Consider a UDP socket sid with no pending errors and bound to local port p_1 . At the head of the socket's receive queue, $rcvq$, is a UDP datagram with source address (i_3, ps_3) and data $data$. Thread tid is blocked in state $\text{RCV2}(sid, n, opts)$.

The blocked call successfully returns $(\mathbf{implode} \ data', \uparrow((i_3, ps_3), b))$. If the number of bytes requested, n , is greater than or equal to the number of bytes of data in the datagram, $data$, then all of $data$ is returned. If n is less than the number of bytes in the datagram, then the first n bytes of data are returned.

A τ transition is made, leaving the thread state $\text{RET}(\text{OK}(\mathbf{implode} \ data', \uparrow((i_3, ps_3), b)))$. If the MSG_PEEK flag was set in $opts$ then the datagram stays on the head of the socket's receive queue; otherwise, it is discarded from the receive queue.

Variations

| | |
|-------|--|
| WinXP | As above, except the number of bytes of data requested n , must be greater than or equal to the length in bytes of $data$. The boolean b equals \mathbf{T} , indicating that all of the datagram's data was read. |
|-------|--|

recv_16 udp: fast fail Fail with EOPNOTSUPP: MSG_WAITALL flag not supported on WinXP, or MSG_OOB flag not supported on FreeBSD and WinXP

$$\begin{aligned}
& h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
& socks := socks \oplus \\
& \quad [(sid, sock \langle pr := \text{UDP_PROTO}(udp) \rangle)] \rangle \\
\text{tid.recv}(fd, n_0, opts_0) \rightarrow & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EOPNOTSUPP}))_{\text{sched_timer}}); \\
& socks := socks \oplus \\
& \quad [(sid, sock \langle pr := \text{UDP_PROTO}(udp) \rangle)] \rangle
\end{aligned}$$

$$\begin{aligned}
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fd = h.fds[fd] \wedge \\
& h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
& opts = \mathbf{list_to_set} \ opts_0 \wedge \\
& ((\text{MSG_OOB} \in \text{opts} \wedge \neg(\text{linux_arch} \ h.\text{arch})) \vee (\text{MSG_WAITALL} \in \text{opts} \wedge \text{windows_arch} \ h.\text{arch}))
\end{aligned}$$

Description

Consider a UDP socket sid referenced by fd . From thread tid , in the RUN state, a $\text{recv}(fd, n_0, opts_0)$ call is made. The MSG_OOB or MSG_WAITALL flags are set in $opts_0$. The call fails with an EOPNOTSUPP error.

A $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EOPNOTSUPP})$.

Model details

The $opts_0$ argument is of type list. In the model it is converted to a set $opts$ using `list_to_set`.

Variations

| | |
|---------|--|
| Posix | As above, except the rule only applies when MSG_OOB is set in $opts_0$. |
| FreeBSD | As above, except the rule only applies when MSG_OOB is set in $opts_0$. |
| Linux | This rule does not apply. |

$recv_{17}$ udp: rc Socket shutdown for reading: fail with ESHUTDOWN on WinXP or succeed on Linux and FreeBSD

$$\frac{h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{socks} := \text{socks} \oplus [(sid, sock \llbracket \text{cantrcvmore} := \mathbf{T}; pr := \text{UDP_Sock}(rcvq) \rrbracket)] \rrbracket}{tid \cdot \text{recv}(fd, n_0, opts_0)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched_timer}}); \text{socks} := \text{socks} \oplus [(sid, sock \llbracket \text{cantrcvmore} := \mathbf{T}; pr := \text{UDP_Sock}(rcvq) \rrbracket)] \rrbracket$$

```

fd ∈ dom(h.fds) ∧
fid = h.fds[fd] ∧
h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
if windows_arch h.arch then ret = FAIL (ESHUTDOWN) ∧ rc = FAST FAIL
else if bsd_arch h.arch then ret = OK("", ↑((*,*), b)) ∧ rc = FAST SUCCEED ∧
sock.es = *
else if linux_arch h.arch then
  rcvq = [] ∧ ret = OK("", ↑((*,*), b)) ∧ rc = FAST SUCCEED ∧ sock.es = *
else ASSERTION_FAILURE"recv_17"

```

Description

Consider a UDP socket sid , referenced by fd , that has been shutdown for reading. From thread tid , which is in the RUN state, a $\text{recv}(fd, n_0, opts_0)$ call is made. On FreeBSD and Linux, if the socket has no pending error the call is successfully, returning $(\text{"", } \uparrow((*,*), b))$; on WinXP the call fails with an ESHUTDOWN error.

A $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{"", } \uparrow((*,*), b)))$ on FreeBSD and Linux, or $\text{RET}(\text{FAIL ESHUTDOWN})$ on WinXP.

Variations

| | |
|---------|---|
| FreeBSD | As above: the call succeeds. |
| Linux | As above: the call succeeds with the additional condition that the socket has an empty receive queue. |
| WinXP | As above: the call fails with an ESHUTDOWN error. |

recv_20 **udp: rc** Successful partial read of datagram on head of socket's receive queue on WinXP

$$\frac{h \langle ts := ts \oplus (tid \mapsto (t)_d); \text{socks} := \text{socks} \oplus [(sid, sock \langle pr := \text{UDP_Sock}(recv) \rangle)] \rangle}{\text{lbl} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } data', \uparrow((i_3, ps_3), \mathbf{F})))_{\text{sched_timer}})) \rangle; \text{socks} := \text{socks} \oplus [(sid, sock)] \rangle}$$

$\text{windows_arch } h.\text{arch} \wedge$
 $\text{recv} = (\text{DGRAM_MSG}(\langle is := i_3; ps := ps_3; data := data \rangle)) :: \text{recv}'' \wedge$
 $\text{sock} = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_Sock}(recv')) \wedge$
 $(\exists fd \text{ ff } n \ n_0 \ \text{opts}_0.$
 $\quad fd \in \text{dom}(h.\text{fds}) \wedge$
 $\quad fid = h.\text{fds}[fd] \wedge$
 $\quad h.\text{files}[fid] = \text{FILE}(\text{FT_SOCKET}(sid), \text{ff}) \wedge$
 $\quad (recv' = \text{if MSG_PEEK} \in (\text{list_to_set } \text{opts}_0) \text{ then } \text{recv} \text{ else } \text{recv}'') \wedge$
 $\quad n = \text{clip_int_to_num } n_0 \wedge$
 $\quad n < \text{length } data \wedge$
 $\quad data' = \text{TAKE } n \ data \wedge$
 $\quad t = \text{RUN} \wedge$
 $\quad rc = \text{FAST SUCCEED} \wedge$
 $\quad \text{lbl} = \text{tid} \cdot \text{recv}(fd, n_0, \text{opts}_0)) \vee$
 $(\exists n \ \text{opts}.$
 $\quad \text{lbl} = \tau \wedge$
 $\quad t = \text{RCV2}(sid, n, \text{opts}) \wedge$
 $\quad rc = \text{SLOW urgent SUCCEED} \wedge$
 $\quad data' = \text{TAKE } n \ data \wedge$
 $\quad n < \text{length } data \wedge$
 $\quad \text{recv}' = \text{if MSG_PEEK} \in \text{opts} \text{ then } \text{recv} \text{ else } \text{recv}'')$

Description

On WinXP, consider a UDP socket sid bound to a local port p_1 and with no pending errors. At the head of the socket's receive queue is a datagram with source address $is := i_3; ps := ps_3$ and data $data$. This rule covers two cases:

In the first, from thread tid , which is in the RUN state, a $\text{recv}(fd, n_0, \text{opts}_0)$ call is made where fd refers to the socket sid . The amount of data to be read, n_0 bytes, is less than the number of bytes of data in the datagram, $data$. The call successfully returns the first n_0 bytes of data from the datagram, $data'$. A $\text{tid} \cdot \text{recv}(fd, n_0, \text{opts}_0)$ transition is made leaving the thread state $\text{RET}(\text{OK}(\text{implode } data', \uparrow((i_3, ps_3), \mathbf{F})))$ where the \mathbf{F} indicates that not all of the datagram's data was read. The datagram is discarded from the socket's receive queue unless the MSG_PEEK flag was set in opts_0 , in which case the whole datagram remains on the socket's receive queue.

In the second case, thread tid is blocked in state $\text{RCV2}(sid, n, \text{opts})$ where the number of bytes to be read, n , is less than the number of bytes of data in the datagram. There is now data to be read so a τ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{implode } data', \uparrow((i_3, ps_3), \mathbf{F})))$ where the \mathbf{F} indicated that not all of the datagram's data was read. The datagram is discarded from the socket's receive queue unless the MSG_PEEK flag was set in opts , in which case the whole datagram remains on the socket's receive queue.

Model details

The amount of data requested, n_0 , is clipped to a natural number from an integer, using `clip_int_to_num`. POSIX specifies an unsigned type for n_0 and this is one possible model thereof.

The data itself is represented as a *byte* list in the datagram but is returned a *string*, so the `implode` function is used to do the conversion.

In the model the return value is $\text{OK}(\text{implode } data', \uparrow((i_3, ps_3), \mathbf{F}))$ where the \mathbf{F} represents not all the data in the datagram at the head of the socket's receive queue being read. What actually happens is that an EMSGSIZE error is returned, and the data is put into the read buffer specified when the `recv()` call was made.

Variations

| | |
|---------|---------------------------|
| Posix | This rule does not apply. |
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

recv_21 **udp: fast succeed** Read zero bytes of data from an empty receive queue on FreeBSD

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 \text{socks} := \text{socks} \oplus \\
 \quad [(sid, \text{sock} \langle pr := \text{UDP_Sock}([\] \rangle)] \\
 \hline
 \text{tid} \cdot \text{recv}(fd, n_0, \text{opts}_0) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{""}, \uparrow((*, *), b))))_{\text{sched_timer}}); \\
 \text{socks} := \text{socks} \oplus \\
 \quad [(sid, \text{sock} \langle pr := \text{UDP_Sock}([\] \rangle)]
 \end{array}$$

bsd_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $0 = \text{clip_int_to_num } n_0$

Description

On FreeBSD, consider a UDP socket sid , referenced by fd , with an empty receive queue. From thread tid , which is in the RUN state, a $\text{recv}(fd, n_0, \text{opts}_0)$ call is made where $n_0 = 0$. The call succeeds, returning the empty string and not specifying an address: $\text{OK}(\text{""}, \uparrow((*, *), b))$.

A $\text{tid} \cdot \text{recv}(fd, n_0, \text{opts}_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{""}, \uparrow((*, *), b)))$.

Variations

| | |
|-------|---|
| Posix | This rule does not apply: see rules <i>recv_12</i> and <i>recv_13</i> . |
| Linux | This rule does not apply: see rules <i>recv_12</i> and <i>recv_13</i> . |
| WinXP | This rule does not apply: see rules <i>recv_12</i> and <i>recv_13</i> . |

recv_22 **udp: fast fail** Fail with EINVAL on WinXP: socket is unbound

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 \text{socks} := \text{socks} \oplus \\
 \quad [(sid, \text{sock} \langle ps_1 := *; pr := \text{UDP_PROTO}(udp) \rangle)] \\
 \hline
 \text{tid} \cdot \text{recv}(fd, n_0, \text{opts}_0) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVALID}))_{\text{sched_timer}}); \\
 \text{socks} := \text{socks} \oplus \\
 \quad [(sid, \text{sock} \langle ps_1 := *; pr := \text{UDP_PROTO}(udp) \rangle)]
 \end{array}$$

windows_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$

$$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$$

Description

On WinXP, consider a UDP socket sid referenced by fd that is not bound to a local port. A $\text{recv}(fd, n_0, opts_0)$ call is made from thread tid which is in the RUN state. The call fails with an EINVAL error.

A $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EINVAL})$.

Variations

| | |
|---------|---------------------------|
| Posix | This rule does not apply. |
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

recv_23 udp: rc Read ICMP error from receive queue and fail with that error on WinXP

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (t)_d); \\
 socks := socks \oplus \\
 [(sid, sock \langle pr := \text{UDP_Sock}(recvq) \rangle)] \rangle
 \end{array}
 \xrightarrow{lbl}
 \begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}); \\
 socks := socks \oplus \\
 [(sid, sock \langle pr := \text{UDP_Sock}(recvq') \rangle)] \rangle
 \end{array}$$

$$\begin{array}{l}
 \text{windows_arch } h.arch \wedge \\
 recvq = (\text{DGRAM_ERROR}(\langle e := err \rangle)) :: recvq' \wedge \\
 ((\exists fd \ n_0 \ opts_0 \ fid \ ff.t = \text{RUN} \wedge \\
 \quad lbl = tid \cdot \text{recv}(fd, n_0, opts_0) \wedge \\
 \quad rc = \text{FAST FAIL} \wedge \\
 \quad fd \in \text{dom}(h.fds) \wedge \\
 \quad fid = h.fds[fd] \wedge \\
 \quad h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff)) \vee \\
 (\exists n \ opts.t = \text{RCV2}(sid, n, opts) \wedge \\
 \quad lbl = \tau \wedge \\
 \quad rc = \text{SLOW urgent FAIL}))
 \end{array}$$

Description

On WinXP, consider a UDP socket sid referenced by fd . At the head of the socket's receive queue, $recvq$, is an ICMP message with error err . This rule covers two cases.

In the first, thread tid is in the RUN state and a $\text{recv}(fd, n_0, opts_0)$ call is made. The call fails with error err , making a $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition. This leaves the thread state $\text{RET}(\text{FAIL } err)$, and the socket with the ICMP message removed from its receive queue.

In the second case, thread tid is blocked in state $\text{RCV2}(sid, n_0, opts_0)$. A τ transition is made, leaving the thread state $\text{RET}(\text{FAIL } err)$, and the socket with the ICMP message removed from its receive queue.

Variations

| | |
|---------|---------------------------|
| Posix | This rule does not apply. |
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

recv_24 **udp: fast fail** Fail with pending error

$$h \{ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$socks := socks \oplus$$

$$[(sid, \text{SOCK}(\uparrow fd, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, \uparrow e, cantsndmore, cantrcvmore, \text{UDP_PROTO}(udp)))]\}$$

$\frac{tid \cdot \text{recv}(fd, n_0, opts_0)}{}$

$$h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}});$$

$$socks := socks \oplus$$

$$[(sid, \text{SOCK}(\uparrow fd, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{UDP_PROTO}(udp)))]\}$$

$fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $opts = \text{list_to_set } opts_0 \wedge$
 $(\neg \text{linux_arch } h.arch \implies \exists p_2.ps_2 = \uparrow p_2) \wedge$
 $es = \text{if MSG_PEEK} \in opts \text{ then } \uparrow e \text{ else } *$

Description

From thread *tid*, which is in the RUN state, a `recv(fd, n0, opts0)` call is made. *fd* refers to a UDP socket that has local address ($\uparrow i_1, \uparrow p_1$), has its peer port set: $ps_2 = \uparrow p_2$, and has pending error $\uparrow e$.

The call fails returning the pending error: a `tid·recv(fd, n0, opts0)` transition is made leaving the thread state `RET(FAIL EAGAIN)`. If the `MSG_PEEK` flag was set in *opts*₀ then the socket's pending error remains, otherwise it is cleared.

Model details

The *opts*₀ argument to `recv()` is of type `msgbflag list`, but it is converted to a set, *opts*, using `list_to_set`.

Variations

| | |
|-------|---|
| Linux | The socket need not have its peer port set. |
|-------|---|

15.21 send() (TCP only)

`send : fd * (ip * port) option * string * msgbflag list → string`

This section describes the behaviour of `send()` for TCP sockets. A call to `send(fd, *, data, flags)` enqueues data on the TCP socket's send queue. Here *fd* is a file descriptor referring to the TCP socket to enqueue data on. The second argument, of type `(ip * port) option`, is the destination address of the data for UDP, but for a TCP socket it should be set to `*` (the socket must be connected to a peer before `send()` can be called). The *data* is the data to be sent. Finally, *flags* is a list of flags for the `send()` call; possible flags are: `MSG_OOB`, specifying that the data to be sent is out-of-band data, and `MSG_DONTWAIT`, specifying that non-blocking behaviour is to be used for this call. The `MSG_WAITALL` and `MSG_PEEK` flags may also be set, but as they are meaningless for `send()` calls, FreeBSD ignores them, and Linux and WinXP fail with `EOPNOTSUPP`. The returned `string` is any data that was not sent.

For a successful `send()` call, the socket must be in a synchronised state, must not be shutdown for writing, and must not have a pending error.

If there is not enough room on a socket's send queue then a `send()` call may block until space becomes available. For a successful blocking `send()` call on FreeBSD the entire string will be enqueued on the socket's send queue.

15.21.1 Errors

In addition to errors returned via ICMP (see `deliver_in_icmp_3` (p337)), a call to `send()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|------------|---|
| EAGAIN | Non-blocking send() call would block. |
| ENOTCONN | Socket not connected on FreeBSD and WinXP. |
| EOPNOTSUPP | Message flags MSG_PEEK and MSG_WAITALL not supported. Linux and WinXP. |
| EPIPE | Socket not connected on Linux; or socket shutdown for writing on FreeBSD and Linux. |
| ESHUTDOWN | Socket shutdown for writing on WinXP. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| EINTR | The system was interrupted by a caught signal. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.21.2 Common cases

A TCP socket is created and successfully connects with a peer; data is then sent to the peer: *socket_1; return_1; connect_1; return_1; ... connect_2; return_1; send_1; ...*

15.21.3 API

```
Posix:    ssize_t send(int socket, const void *buffer, size_t length, int flags);
FreeBSD:  ssize_t send(int s, const void *msg, size_t len, int flags);
Linux:    int send(int s, const void *msg, size_t len, int flags);
WinXP:    int send(SOCKET s, const char *buf, int len, int flags);
```

In the Posix interface:

- **socket** is the file descriptor of the socket to send from, corresponding to the `fd` argument of the model `send()`.
- **message** is a pointer to the data to be sent of length `length`. The two together correspond to the `string` argument of the model `send()`.
- **flags** is a disjunction of the message flags for the `send()` call, corresponding to the `msgbflag` list in the model `send()`.
- the returned `ssize_t` is either non-negative or `-1`. If it is non-negative then it is the amount of data from `message` that was sent. If it is `-1` then it indicates an error, in which case the error is stored in `errno`. This corresponds to the model `send()`'s return value of type `string` which is the data that was not sent. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.21.4 Model details

If the call blocks then the thread enters state `SEND2(sid, *, str, opts)` (the optional parameter is used for UDP only), where

- `sid` : `sid` is the identifier of the socket that made the `send()` call,
- `str` : `string` is the data to be sent, and

- *opts* : *msgbflag* list is the set of options for the `send()` call.

The following errors are not modelled:

- In Posix and on all three architectures, `EDESTADDRREQ` indicates that the socket is not connection-mode and no peer address is set. This doesn't apply to TCP, which is a connection-mode protocol.
- In Posix, `EACCES` signifies that write access to the socket is denied. This is not modelled here.
- On FreeBSD and Linux, `EFAULT` signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to `accept()` that is excluded by the clean interface used in the model.
- In Posix and on Linux, `EINVAL` signifies that an invalid argument was passed. The typing of the model interface prevents this from happening.
- In Posix, `EIO` signifies that an I/O error occurred while reading from or writing to the file system. This is not modelled.
- On Linux, `EMSGSIZE` indicates that the message is too large to be sent all at once, as the socket requires; this is not a requirement for TCP sockets.
- In Posix, `ENETDOWN` signifies that the local network interface used to reach the destination is down. This is not modelled.

The following flags are not modelled:

- On Linux, `MSG_CONFIRM` is used to tell the link layer not to probe the neighbour.
- On Linux, `MSG_NOSIGNAL` requests not to send `SIGPIPE` errors on stream-oriented sockets when the other end breaks the connection.
- On FreeBSD and WinXP, `MSG_DONTROUTE` is used by routing programs.
- On FreeBSD, `MSG_EOR` is used to indicate the end of a record for protocols that support this. It is not modelled because TCP does not support records.
- On FreeBSD, `MSG_EOF` is used to implement Transaction TCP which is not modelled here.

15.21.5 Summary

| | | |
|----------------|------------------------------------|---|
| <i>send_1</i> | tcp: fast succeed | Successfully send data without blocking |
| <i>send_2</i> | tcp: block | Block waiting for space in socket's send queue |
| <i>send_3</i> | tcp: slow nonurgent succeed | Successfully return from blocked state having sent data |
| <i>send_3a</i> | tcp: block | From blocked state, transfer some data to the send queue and remain blocked |
| <i>send_4</i> | tcp: fast fail | Fail with <code>EAGAIN</code> : non-blocking semantics requested and call would block |
| <i>send_5</i> | tcp: fast fail | Fail with pending error |
| <i>send_5a</i> | tcp: slow urgent fail | Fail from blocked state with pending error |
| <i>send_6</i> | tcp: fast fail | Fail with <code>ENOTCONN</code> or <code>EPIPE</code> : socket not connected |
| <i>send_7</i> | tcp: rc | Fail with <code>EPIPE</code> or <code>ESHUTDOWN</code> : socket shut down for writing |
| <i>send_8</i> | tcp: fast fail | Fail with <code>EOPNOTSUPP</code> : message flag not valid |

15.21.6 Rules

send_1 **tcp: fast succeed** Successfully send data without blocking

$$h \{ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$

$$socks := socks \oplus$$

$$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore},$$

$$\text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]\}$$

$$\xrightarrow{tid \cdot \text{send}(fd, *, \text{implode } str, opts_0)}$$

$$h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } str''))_{\text{sched_timer}}));$$

$$socks := socks \oplus$$

$$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore},$$

$$\text{TCP_Sock}(st, cb, *, sndq @ str', sndurp', rcvq, rcvurp, iobc)))]\}$$

$$st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge$$

$$opts = \text{list_to_set } opts_0 \wedge$$

$$fd \in \text{dom}(h.fds) \wedge$$

$$fd = h.fds[fd] \wedge$$

$$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$space \in \text{send_queue_space}$$

$$(sf.n(\text{SO_SNDBUF}))(\text{length } sndq)(\text{MSG_OOB} \in opts)h.arch \text{ cb.t_maxseg } i_2 \wedge$$

$$(\{\text{MSG_PEEK}; \text{MSG_WAITALL}\} \cap opts = \emptyset \vee \text{bsd_arch } h.arch) \wedge$$

(if $space \geq \text{length } str$ **then**

$$str' = str \wedge str'' = []$$

else

$$(ff.b(\text{O_NONBLOCK}) \vee (\text{MSG_DONTWAIT} \in opts \wedge \neg \text{bsd_arch } h.arch)) \wedge$$

$$(\text{if } \text{bsd_arch } h.arch \text{ then } space \geq sf.n(\text{SO_SNDLOWAT})$$

$$\text{ else } space > 0) \wedge$$

$$(str', str'') = \text{SPLIT } space \text{ } str$$

) \wedge

$$sndurp' = (\text{if } (\text{MSG_OOB} \in opts) \wedge (n = \text{length } str)$$

$$\text{ then } \uparrow(\text{length}(sndq @ str') - 1)$$

$$\text{ else } sndurp)$$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, *, \text{implode } str, opts_0)$ call is made. fd refers to a TCP socket sid that has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The MSG_PEEK and MSG_WAITALL flags are not set in $opts_0$. $space$ is the space in the socket's send queue, calculated using send_queue_space (p93).

This rule covers two cases: (1) there is space in the socket's send queue for all the data; and (2) there is not space for all the data but the call is non-blocking (the MSG_DONTWAIT flag is set in $opts$ or the socket's O_NONBLOCK flag is set), and the space is greater than zero, or, on FreeBSD, greater than the minimum number of bytes for $\text{send}()$ operations on the socket, $sf.n(\text{SO_SNDLOWAT})$.

In (1) all of the data str is appended to the socket's send queue and the returned string, str'' , is the empty string. In (2), the first $space$ bytes of data, str' , are appended to the socket's send queue and the remaining data, str'' , is returned.

In both cases a $tid \cdot \text{send}(fd, *, \text{implode } str, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{implode } str''))$. If the data was marked as out-of-band, $\text{MSG_OOB} \in opts$, then the socket's send urgent pointer will point to the end of the send queue.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte list* when the datagram is constructed. Here the data, str is of type *byte list* and in the transition $\text{implode } str$ is used to convert it into a string.

The $opts_0$ argument is of type **list**. In the model it is converted to a **set** $opts$ using list_to_set . The presence of MSG_PEEK is checked for in $opts$ rather than in $opts_0$.

Variations

| | |
|---------|--|
| FreeBSD | The MSG_PEEK and MSG_WAITALL flags may be set in $opts_0$ but for the call to be non-blocking the socket's O_NONBLOCK flag must be set: the MSG_DONTWAIT flag has no effect. |
|---------|--|

send_2 **tcp: block** Block waiting for space in socket's send queue

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 socks := socks \oplus \\
 \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
 \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))] \\
 \hline
 tid \cdot \text{send}(fd, *, \mathbf{implode} \ str, opts_0) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_{\text{never_timer}}); \\
 socks := socks \oplus \\
 \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
 \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]
 \end{array}$$

$$\begin{array}{l}
 opts = \mathbf{list_to_set} \ opts_0 \wedge \\
 fd \in \mathbf{dom}(h.fds) \wedge \\
 fid = h.fds[fd] \wedge \\
 h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 \neg((\neg \text{bsd_arch } h.arch \wedge \text{MSG_DONTWAIT} \in opts) \vee ff.b(\text{O_NONBLOCK})) \wedge
 \end{array}$$

$$\begin{array}{l}
 space \in \text{send_queue_space} \\
 \quad (sf.n(\text{SO_SNDBUF}))(\mathbf{length} \ sndq)(\text{MSG_OOB} \in opts)h.arch \ cb.t_maxseg \ i_2 \wedge
 \end{array}$$

$$\{ \text{MSG_PEEK}; \text{MSG_WAITALL} \} \cap opts = \emptyset \vee \text{bsd_arch } h.arch \wedge$$

$$\begin{array}{l}
 ((st \in \{ \text{ESTABLISHED}; \text{CLOSE_WAIT} \} \wedge \\
 \quad space < \mathbf{length} \ str) \vee \\
 (\text{linux_arch } h.arch \wedge st \in \{ \text{SYN_SENT}; \text{SYN_RECEIVED} \}))
 \end{array}$$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, *, \mathbf{implode} \ str, opts_0)$ call is made. fd refers to a TCP socket sid that has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The call is a blocking one: the socket's O_NONBLOCK flag is not set and the MSG_DONTWAIT flag is not set in $opts_0$. The MSG_PEEK and MSG_WAITALL flags are not set in $opts_0$.

The space in the socket's send queue, $space$ (calculated using send_queue_space (p93)), is less than the length in bytes of the data to be sent, str .

The call blocks, leaving the thread state $\text{SEND2}(sid, *, str, opts)$ via a $tid \cdot \text{send}(fd, *, \mathbf{implode} \ str, opts_0)$ transition.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte list* when the datagram is constructed. Here the data, str is of type *byte list* and in the transition $\mathbf{implode} \ str$ is used to convert it into a string.

Variations

| | |
|---------|--|
| FreeBSD | The MSG_PEEK, MSG_WAITALL, and MSG_DONTWAIT flags may all be set in $opts_0$: all three are ignored by FreeBSD. |
| Linux | In addition to the above, the rule also applies if connection establishment is still taking place for the socket: it is in state SYN_SENT or SYN_RECEIVED. |

send_3 **tcp: slow nonurgent succeed** **Successfully return from blocked state having sent data**

$$\begin{aligned}
& h \{ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d); \\
& socks := socks \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]\} \\
\tau \rightarrow & h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } str''))_{\text{sched_timer}}); \\
& socks := socks \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, sndq @ str', sndurp', rcvq, rcvurp, iobc)))]\}
\end{aligned}$$

$$st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge$$

$$space \in \text{send_queue_space} \\ (sf.n(\text{SO_SNDBUF}))(\text{length } sndq)(\text{MSG_OOB} \in opts)h.arch cb.t_maxseg i_2 \wedge$$

$$\begin{aligned}
space & \geq \text{length } str \wedge \\
str' & = str \wedge str'' = [] \wedge \\
sndurp' & = \text{if } \text{MSG_OOB} \in opts \text{ then } \uparrow(\text{length}(sndq @ str') - 1) \\
& \quad \text{else } sndurp
\end{aligned}$$

Description

Thread *tid* is blocked in state $\text{SEND2}(sid, *, str, opts)$ where the TCP socket *sid* has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT .

The space in the socket's send queue, *space* (calculated using send_queue_space (p93)), is greater than or equal to the length of the data to be sent, *str*. The data is appended to the socket's send queue and the call successfully returns the empty string. A τ transition is made, leaving the thread state $\text{RET}(\text{OK}''')$. If the data was marked as out-of-band, $\text{MSG_OOB} \in opts$, then the socket's urgent pointer will be updated to point to the end of the socket's send queue.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte* list when the datagram is constructed. Here the data, *str* is of type *byte* list and in the transition **implode** *str* is used to convert it into a string.

send_3a **tcp: block** **From blocked state, transfer some data to the send queue and remain blocked**

$$\begin{aligned}
& h \{ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d); \\
& socks := socks \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]\} \\
\tau \rightarrow & h \{ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str'', opts))_{\text{never_timer}}); \\
& socks := socks \oplus \\
& \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\
& \quad \quad \text{TCP_Sock}(st, cb, *, sndq @ str', sndurp', rcvq, rcvurp, iobc)))]\}
\end{aligned}$$

$$st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge$$

$$space \in \text{send_queue_space} \\ (sf.n(\text{SO_SNDBUF}))(\text{length } sndq)(\text{MSG_OOB} \in opts)h.arch cb.t_maxseg i_2 \wedge$$

$$\begin{aligned}
space & < \text{length } str \wedge space > 0 \wedge \\
(str', str'') & = \text{SPLIT } space \text{ } str \wedge \\
sndurp' & = \text{if } \text{MSG_OOB} \in opts \text{ then } \uparrow(\text{length}(sndq @ str') - 1) \text{ else } sndurp
\end{aligned}$$

Description

Thread tid is blocked in state $\text{SEND2}(sid, *, str, opts)$ where TCP socket sid has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The amount of space in the socket's send queue, $space$ (calculated using send_queue_space (p93)), is less than the length of the remaining data to be sent, str , and greater than 0. The socket's send queue is filled by appending the first $space$ bytes of str , str' , to it.

A τ transition is made, leaving the thread state $\text{SEND2}(sid, *, str'', opts)$ where str'' is the remaining data to be sent. If the data in str is out-of-band, MSG_OOB is set in $opts$, then the socket's urgent pointer is updated to point to the end of the socket's send queue.

Note it is unclear whether or not MSG_OOB should be removed from $opts$ in the state.

send_4 **tcp: fast fail Fail with EAGAIN: non-blocking semantics requested and call would block**

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{send}(fd, *, \text{implode } str, opts_0)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched_timer}}) \rangle$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \text{cantrcvmore}, \\ &\quad \text{TCP_Sock}(st, cb, *, \text{sndq}, \text{sndurp}, \text{rcvq}, \text{rcvurp}, iobc)) \wedge \\ &opts = \text{list_to_set } opts_0 \wedge \end{aligned}$$

$$(\{\text{MSG_PEEK}; \text{MSG_WAITALL}\} \cap opts = \emptyset \vee \text{bsd_arch } h.arch) \wedge$$

$$((\neg \text{bsd_arch } h.arch \wedge \text{MSG_DONTWAIT} \in opts) \vee ff.b(\text{O_NONBLOCK})) \wedge$$

$$\begin{aligned} &((st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge \\ &\quad space \in \text{send_queue_space} \\ &\quad (sf.n(\text{SO_SNDBUF}))(\text{length } \text{sndq})(\text{MSG_OOB} \in opts)h.arch cb.t_maxseg i_2 \wedge \\ &\quad \neg(space \geq \text{length } str \vee (\mathbf{if } \text{bsd_arch } h.arch \mathbf{then } space \geq sf.n(\text{SO_SNDLOWAT}) \mathbf{else } space > 0))) \vee \\ &(st \in \{\text{SYN_SENT}; \text{SYN_RECEIVED}\} \wedge \\ &\quad \text{linux_arch } h.arch)) \end{aligned}$$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, *, \text{implode } str, opts_0)$ call is made. fd refers to a TCP socket that has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The call is a non-blocking one: either the socket's O_NONBLOCK flag is set or the MSG_DONTWAIT flag is set in $opts_0$. The MSG_PEEK and MSG_WAITALL flags are not set in $opts_0$.

The space in the socket's send queue, $space$ (calculated using send_queue_space (p93)), is less than both the length of the data to send str ; and on FreeBSD is less than the minimum number of bytes for socket send operations, $sf.n(\text{SO_SNDLOWAT})$, or on Linux and WinXP is equal to zero. The call would have to block, but because it is non-blocking, it fails with an EAGAIN error.

A $tid \cdot \text{send}(fd, *, \text{implode } str, opts_0)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL EAGAIN})$.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte list* when the datagram is constructed. Here the data, str is of type *byte list* and in the transition $\text{implode } str$ is used to convert it into a string.

The $opts_0$ argument is of type **list**. In the model it is converted to a **set** $opts$ using list_to_set . The presence of MSG_PEEK is checked for in $opts$ rather than in $opts_0$.

Variations

| | |
|---------|--|
| FreeBSD | For the call to be non-blocking, the socket's O_NONBLOCK flag must be set; the MSG_DONTWAIT flag is ignored. Additionally, the MSG_PEEK and MSG_WAITALL flags may be set in $opts_0$ as they are also ignored. |
| Linux | This rule also applies if the socket is in state SYN_SENT or SYN_RECEIVED, in which case the send queue size does not matter. |

send_5 tcp: fast fail Fail with pending error

$$\begin{array}{l}
h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle; \\
socks := socks \oplus \\
\quad [(sid, sock \langle es := \uparrow e \rangle)] \\
\hline
tid \cdot \text{send}(fd, addr, \mathbf{implode} \ str, opts_0) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rangle; \\
\quad socks := socks \oplus \\
\quad \quad [(sid, sock \langle es := * \rangle)]
\end{array}$$

$$\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
fd = h.fds[fd] \wedge \\
h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
\text{proto_of } sock.pr = \text{PROTO_TCP}
\end{array}$$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, addr, \mathbf{implode} \ str, opts_0)$ call is made. fd refers to a socket $sock$ identified by sid with pending error $\uparrow e$. The call fails, returning the pending error.

A $tid \cdot \text{send}(fd, addr, \mathbf{implode} \ str, opts)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL } e)$.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte* list when the datagram is constructed. Here the data, str is of type *byte* list and in the transition $\mathbf{implode} \ str$ is used to convert it into a string.

send_5a tcp: slow urgent fail Fail from blocked state with pending error

$$\begin{array}{l}
h \langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d) \rangle; \quad \tau \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rangle; \\
socks := socks \oplus \quad \quad \quad socks := socks \oplus \\
\quad [(sid, sock \langle es := \uparrow e \rangle)] \quad \quad \quad [(sid, sock \langle es := * \rangle)]
\end{array}$$

$$\text{proto_of } sock.pr = \text{PROTO_TCP}$$

Description

Thread tid is blocked in state $\text{SEND2}(sid, *, str, opts)$ from an earlier $\text{send}()$ call. The TCP socket sid has pending error $\uparrow e$ so the call can now return, failing with the error.

A τ transition is made, leaving the thread state $\text{RET}(\text{FAIL } e)$.

send_6 tcp: fast fail Fail with ENOTCONN or EPIPE: socket not connected

$$\begin{array}{l}
h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \\
tid \cdot \text{send}(fd, *, \mathbf{implode} \ str, opts_0) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}) \rangle
\end{array}$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

```

fid = h.fds[fd] ∧
h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
sock = (h.socks[sid]) ∧
TCP_PROTO(tcp_sock) = sock.pr ∧
sock.es = * ∧
(tcp_sock.st ∈ {CLOSED; LISTEN}) ∨

(tcp_sock.st ∈ {SYN_SENT; SYN_RECEIVED}) ∧ ¬(linux_arch h.arch) ∨
F (* Placeholder for: if tcp_disconnect or tcp_usrclose has been invoked *)
) ∧
err = (if linux_arch h.arch then EPIPE else ENOTCONN)

```

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, *, \text{implode } str, \text{opts}_0)$ call is made. fd refers to a TCP socket $sock$ identified by sid that does not have a pending error. The socket is not synchronised: it is in state CLOSED, LISTEN, SYN_SENT, or SYN_RECEIVED. The call fails with an ENOTCONN error, or EPIPE on Linux.

A $tid.\text{send}(fd, *, \text{implode } str, \text{opts}_0)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL } err)$ where err is one of the above errors.

Model details

The data to be sent is of type `string` in the $\text{send}()$ call but is a `byte` list when the datagram is constructed. Here the data, str is of type `byte` list and in the transition $\text{implode } str$ is used to convert it into a string.

Variations

| | |
|-------|---|
| Linux | The rule does not apply if the socket is in state SYN_RECEIVED or SYN_SENT. |
|-------|---|

send_7 **tcp: rc** Fail with EPIPE or ESHUTDOWN: socket shut down for writing

```

h (ts := ts ⊕ (tid ↦ (t)d);
socks := socks ⊕
  [(sid, SOCK(↑ fid, sf, is1, ps1, is2, ps2, es, T, cantrcvmore, TCP_PROTO(tcp)))]])
 $\xrightarrow{lbl}$  h (ts := ts ⊕ (tid ↦ (RET(FAIL err))sched_timer);
socks := socks ⊕
  [(sid, SOCK(↑ fid, sf, is1, ps1, is2, ps2, es, T, cantrcvmore, TCP_PROTO(tcp)))]])

( ( ( ( ∃ fd ff str opts0 i2 p2.
  fd ∈ dom(h.fds) ∧
  fid = h.fds[fd] ∧
  h.files[fd] = FILE(FT_SOCKET(sid), ff) ∧
  t = RUN ∧
  lbl = tid.send(fd, *, implode str, opts0) ∧
  rc = FAST FAIL ∧
  is2 = ↑ i2 ∧ ps2 = ↑ p2 ∧
  (if tcp.st ≠ CLOSED then
    ∃ i1 p1. is1 = ↑ i1 ∧ ps1 = ↑ p1
  else T)
) ) ) ∨ ( ( ∃ opts str.
  t = SEND2(sid, *, str, opts) ∧
  lbl = τ ∧
  rc = SLOW urgent FAIL
) ) ) ∧

(if windows_arch h.arch then err = ESHUTDOWN
else err = EPIPE)

```

Description

This rule covers two cases: (1) from thread tid , which is in the RUN state, a $\text{send}(fd, *, \mathbf{implode} \text{ } str, opts_0)$ call is made; and (2) thread tid is blocked in state $\text{SEND2}(sid, *, str, opts)$. In (1), fd refers to a TCP socket sid that has binding quad $(is_1, ps_1, \uparrow i_2, \uparrow p_2)$. In both cases the socket is shutdown for writing. The call fails with an EPIPE error.

The thread is left in state $\text{RET}(\text{FAIL EPIPE})$, via a $tid \cdot \text{send}(fd, *, \mathbf{implode} \text{ } str, opts_0)$ transition in (1) or a τ transition in (2).

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte list* when the datagram is constructed. Here the data, str is of type *byte list* and in the transition $\mathbf{implode} \text{ } str$ is used to convert it into a string.

Variations

| | |
|-------|--|
| WinXP | The call fails with an ESHUTDOWN error instead of EPIPE. |
|-------|--|

$send_8$ **tcp: fast fail** Fail with EOPNOTSUPP: message flag not valid

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{send}(fd, *, \mathbf{implode} \text{ } str, opts_0)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EOPNOTSUPP}))_{\text{sched_timer}}) \rangle$$

$fd \in \text{dom}(h.fds) \wedge$
 $fd = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $\text{proto_of}(h.socks[sid]).pr = \text{PROTO_TCP} \wedge$
 $opts = \mathbf{list_to_set} \text{ } opts_0 \wedge$
 $(\text{MSG_PEEK} \in opts \vee \text{MSG_WAITALL} \in opts) \wedge$
 $\neg \text{bsd_arch } h.arch$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, *, \mathbf{implode} \text{ } str, opts_0)$ call is made. fd refers to a TCP socket identified by sid . Either the MSG_PEEK or MSG_WAITALL flag is set in $opts_0$. These flags are not supported so the call fails with an EOPNOTSUPP error.

A $tid \cdot \text{send}(fd, *, \mathbf{implode} \text{ } str, opts_0)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL EOPNOTSUPP})$.

Model details

The data to be sent is of type **string** in the $\text{send}()$ call but is a *byte list* when the datagram is constructed. Here the data, str is of type *byte list* and in the transition $\mathbf{implode} \text{ } str$ is used to convert it into a string.

The $opts_0$ argument is of type **list**. In the model it is converted to a **set** $opts$ using $\mathbf{list_to_set}$. The presence of MSG_PEEK is checked for in $opts$ rather than in $opts_0$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
|---------|---------------------------|

15.22 send() (UDP only)

$\text{send} : (fd * (ip * port) \text{ option } * \text{string } * \text{msgbflag list}) \rightarrow \text{string}$

This section describes the behaviour of $\text{send}()$ for UDP sockets. A call to $\text{send}(fd, addr, data, flags)$ enqueues a UDP datagram to send to a peer. Here the fd argument is a file descriptor referring to a UDP socket from

which to send data. The destination address of the data can be specified either by the *addr* argument, which can be $\uparrow(i_3, p_3)$ or $*$, or by the socket's peer address (its *is₂* and *ps₂* fields) if set. For a successful `send()`, at least one of these two must be specified. If the socket has a peer address set and *addr* is set to $\uparrow(i_3, p_3)$, then the address used is architecture-dependent: on FreeBSD the `send()` call will fail with an EISCONN error; on Linux and WinXP *i₃*, *p₃* will be used.

The string, *data*, is the data to be sent. The length in bytes of *data* must be less than the architecture-dependent maximum payload for a UDP datagram. Sending a string of length zero bytes is acceptable.

The `msgbflag` list is the list of message flags for the `send()` call. The possible flags are MSG_DONTWAIT and MSG_OOB. MSG_DONTWAIT specifies that non-blocking behaviour should be used for this call: see rules *send_10* and *send_11*. MSG_OOB specifies that the data to be sent is out-of-band data, which is not meaningful for UDP sockets. FreeBSD ignores this flag, but on Linux and WinXP the `send()` call will fail: see rule *send_20*.

The return value of the `send()` call is a string of the data which was not sent. A partial send may occur when the call is interrupted by a signal after having sent some data.

For a datagram to be sent, the socket must be bound to a local port. When a `send()` call is made, the socket is autobound to an ephemeral port if it does not have its local port bound.

A successful `send()` call only guarantees that the datagram has been placed on the host's out queue. It does not imply that the datagram has left the host, let alone been successfully delivered to its destination.

A call to `send()` may block if there is no room on the socket's send buffer and non-blocking behaviour has not been requested.

15.22.1 Errors

In addition to errors returned via ICMP (see *deliver_in_icmp_3* (p337)), a call to `send()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|---------------|---|
| EADDRINUSE | The socket's peer address is not set and the destination address specified would give the socket a binding quad i_1, p_1, i_2, p_2 which is already in use by another socket. |
| EADDRNOTAVAIL | There are no ephemeral ports left for autobinding to. |
| EAGAIN | The <code>send()</code> call would block and non-blocking behaviour is requested. This may have been done either via the MSG_DONTWAIT flag being set in the <code>send()</code> flags or the socket's O_NONBLOCK flag being set. |
| EDESTADDRREQ | The socket does not have its peer address set, and no destination address was specified. |
| EINTR | A signal interrupted <code>send()</code> before any data was transmitted. |
| EISCONN | On FreeBSD, a destination address was specified and the socket has a peer address set. |
| EMSGSIZE | The message is too large to be sent in one datagram. |
| ENOTCONN | The socket does not have its peer address set, and no destination address was specified. This can occur either when the call is first made, or if it blocks and if the peer address is unset by a call to <code>disconnect()</code> whilst blocked. |
| EOPNOTSUPP | The MSG_OOB flag is set on Linux or WinXP. |
| EPIPE | Socket shut down for writing. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

| | |
|---------|-------------------|
| ENOBUFS | Out of resources. |
| ENOMEM | Out of resources. |

15.22.2 Common cases

send_9; return_1;

15.22.3 API

```
Posix:    ssize_t sendto(int socket, const void *message, size_t length,
                    int flags, const struct sockaddr *dest_addr,
                    socklen_t dest_len);
FreeBSD:  ssize_t sendto(int s, const void *msg, size_t len, int flags,
                    const struct sockaddr *to, socklen_t tolen);
Linux:    int sendto(int s, const void *msg, size_t len, int flags,
                    const struct sockaddr *to, socklen_t tolen);
WinXP:    int sendto(SOCKET s, const char* buf, int len, int flags,
                    const struct sockaddr* to, int tolen);
```

In the Posix interface:

- **socket** is the file descriptor of the socket to send from, corresponding to the **fd** argument of the model **send()**.
- **message** is a pointer to the data to be sent of length **length**. The two together correspond to the **string** argument of the model **send()**.
- **flags** is an OR of the message flags for the **send()** call, corresponding to the **msgbflag** list in the model **send()**.
- **dest_addr** and **dest_len** correspond to the **addr** argument of the model **send()**. **dest_addr** is either null or a pointer to a **sockaddr** structure containing the destination address for the data. If it is null it corresponds to **addr = ***. If it contains an address, then it corresponds to **addr = ↑(*i*₃, *p*₃)** where *i*₃ and *p*₃ are the IP address and port specified in the **sockaddr** structure.
- the returned **ssize_t** is either non-negative or -1. If it is non-negative then it is the amount of data from **message** that was sent. If it is -1 then it indicates an error, in which case the error is stored in **errno**. This is different to the model **send()**'s return value of type **string** which is the data that was not sent. On WinXP an error is indicated by a return value of **SOCKET_ERROR**, not -1, with the actual error code available through a call to **WSAGetLastError()**.

There are other functions used to send data on a socket. **send()** is similar to **sendto()** except it does not have the **address** and **address_len** arguments. It is used when the destination address of the data does not need to be specified. **sendmsg()**, another output function, is a more general form of **sendto()**.

15.22.4 Model details

If the call blocks then the thread enters state **SEND2(sid, ↑(addr, *i*₁, *ps*₁, *i*₂, *ps*₂), str, opts)** where

- **sid** : **sid** is the identifier of the socket that made the **send()** call,
- **addr** : (**ip * port**) option is the destination address specified in the **send()** call,
- ***i*₁** : **ip** option is the socket's local IP address, possibly *,
- ***ps*₁** : **port** option is the socket's local port, possibly *,
- ***i*₂** : **ip** option is the IP address of the socket's peer, possibly *,
- ***ps*₂** : **ip** option is the port of the socket's peer, possibly *,

- *str* : string is the data to be sent, and
- *opts* : msgbflag list is the set of options for the send() call.

The following errors are not modelled:

- On FreeBSD, EACCES signifies that the destination address is a broadcast address and the SO_BROADCAST flag has not been set on the socket. Broadcast is not modelled here.
- In Posix, EACCES signifies that write access to the socket is denied. This is not modelled here.
- On FreeBSD and Linux, EFAULT signifies that the pointers passed as either the *address* or *address_len* arguments were inaccessible. This is an artefact of the C interface to accept() that is excluded by the clean interface used in the model.
- In Posix and on Linux, EINVAL signifies that an invalid argument was passed. The typing of the model interface prevents this from happening.
- In Posix, EIO signifies that an I/O error occurred while reading from or writing to the file system. This is not modelled.
- In Posix, ENETDOWN signifies that the local network interface used to reach the destination is down. This is not modelled.

The following flags are not modelled:

- On Linux, MSG_CONFIRM is used to tell the link layer not to probe the neighbour.
- On Linux, MSG_NOSIGNAL requests not to send SIGPIPE errors on stream-oriented sockets when the other end breaks the connection. UDP is not stream-oriented.
- On FreeBSD and WinXP, MSG_DONTROUTE is used by routing programs.
- On FreeBSD, MSG_EOR is used to indicate the end of a record for protocols that support this. It is not modelled because UDP does not support records.
- On FreeBSD, MSG_EOF is used to implement Transaction TCP.

15.22.5 Summary

| | | |
|----------------|---------------------------------|---|
| <i>send_9</i> | udp: fast succeed | Enqueue datagram and return successfully |
| <i>send_10</i> | udp: block | Block waiting to enqueue datagram |
| <i>send_11</i> | udp: fast fail | Fail with EAGAIN: call would block and non-blocking behaviour has been requested |
| <i>send_12</i> | udp: fast fail | Fail with ENOTCONN: no peer address set in socket and no destination address provided |
| <i>send_13</i> | udp: fast fail | Fail with EMSGSIZE: string to be sent is bigger than UDPpayloadMax |
| <i>send_14</i> | udp: fast fail | Fail with EAGAIN, EADDRNOTAVAIL or ENOBUFS: there are no ephemeral ports left |
| <i>send_15</i> | udp: slow urgent succeed | Return from blocked state after datagram enqueued |
| <i>send_16</i> | udp: slow urgent fail | Fail: blocked socket has entered an error state |
| <i>send_17</i> | udp: slow urgent fail | Fail with EMSGSIZE or ENOTCONN: blocked socket has had peer address unset or string to be sent is too big |
| <i>send_18</i> | udp: fast fail | Fail with EOPNOTSUPP: MSG_PEEK flag not supported for send() calls on WinXP; or MSG_OOB flag not supported on WinXP and Linux |
| <i>send_19</i> | udp: fast fail | Fail with EADDRINUSE: on FreeBSD, local and destination address quad in use by another socket |
| <i>send_21</i> | udp: fast fail | Fail with EISCONN: socket has peer address set and destination address is specified in call on FreeBSD |
| <i>send_22</i> | udp: fast fail | Fail with EPIPE or ESHUTDOWN: socket shut down for writing |
| <i>send_23</i> | udp: fast fail | Fail with pending error |

15.22.6 Rules

| | |
|---|---|
| $send_9$ | <p style="text-align: center;">udp: fast succeed Enqueue datagram and return successfully</p> $\frac{h_0 \quad tid.send(fd, addr, \mathbf{implode} \ str, opts_0)}{h \langle ts := ts \oplus (tid \mapsto (RET(OK(“”)))_{sched_timer});$ $socks := socks \oplus$ $\langle [sid, sock \langle es := es; ps_1 := \uparrow p'_1; pr := UDP_PROTO(udp) \rangle \rangle];$ $bound := bound;$ $oq := oq' \rangle}$ |
| $h_0 = h \langle ts := ts \oplus (tid \mapsto (RUN)_d);$ $socks := socks \oplus$ $\langle [sid, sock \langle es := es; pr := UDP_PROTO(udp) \rangle \rangle \rangle \wedge$ $fd \in \mathbf{dom}(h_0.fds) \wedge$ $fd = h_0.fds[fd] \wedge$ $h_0.files[fd] = \mathbf{FILE}(\mathbf{FT_SOCKET}(sid), ff) \wedge$ $sock.cantsndmore = \mathbf{F} \wedge$ $STRLEN(\mathbf{implode} \ str) \leq \mathbf{UDPpayloadMax} \ h_0.arch \wedge$ $((addr \neq *) \vee (sock.is_2 \neq *)) \wedge$ $p'_1 \in \mathbf{autobind}(sock.ps_1, \mathbf{PROTO_UDP}, h_0.socks) \wedge$ $(\mathbf{if} \ sock.ps_1 = * \ \mathbf{then} \ bound = sid :: h_0.bound \ \mathbf{else} \ bound = h_0.bound) \wedge$ $\mathbf{dosend}(h.ifds, h.rttab, (addr, str), (sock.is_1, \uparrow p'_1, sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{T}) \wedge$ $(\mathbf{if} \ \mathbf{bsd_arch} \ h.arch \ \mathbf{then} \ (h_0.socks[sid]).sf.n(\mathbf{SO_SNDBUF}) \geq STRLEN(\mathbf{implode} \ str)$ $\ \mathbf{else} \ \mathbf{MSG_OOB} \notin (\mathbf{list_to_set} \ opts_0)) \wedge$ $\neg(\mathbf{windows_arch} \ h.arch) \implies es = *)$ | |

Description

Consider a UDP socket sid referenced by fd that is not shutdown for writing and has no pending errors. From thread tid , which is in the \mathbf{RUN} state, a call $\mathbf{send}(fd, addr, \mathbf{implode} \ str, opts_0)$ succeeds if:

- the length of str is less than $\mathbf{UDPpayloadMax}$ (p70), the architecture-dependent maximum payload for a UDP datagram.
- The socket has a peer IP address set in its is_2 field or the $addr$ argument is $\uparrow(i_3, p_3)$, specifying a destination address.
- The socket is bound to a local port p'_1 , or it can be autobound to p'_1 and sid added to the list of bound sockets.
- A UDP datagram is constructed from the socket's binding quad $(sock.is_1, \uparrow p'_1, sock.is_2, sock.ps_2)$, the destination address argument $addr$, and the data str . This datagram is successfully enqueued on the outqueue of the host, oq to form outqueue oq' using auxiliary function \mathbf{dosend} (p96).

A $tid.send(fd, addr, \mathbf{implode} \ str, opts_0)$ transition is made, leaving the thread in state $\mathbf{RET}(OK(“”))$ and the host with new outqueue oq' . If the socket was autobound to a port then sid is appended to the host's list of bound sockets.

Model details

The data to be sent is of type \mathbf{string} in the $\mathbf{send}()$ call but is a \mathbf{byte} list when the datagram is constructed. Here the data, str is of type \mathbf{byte} list and in the transition $\mathbf{implode} \ str$ is used to convert it into a string.

Variations

| | |
|-------|---|
| Posix | The $\mathbf{MSG_OOB}$ flag is not set in $opts_0$. |
|-------|---|

| | |
|---------|--|
| FreeBSD | On FreeBSD there is an additional condition for a successful <code>send()</code> : the amount of data to be sent must be less than or equal to the size of the socket's send buffer. |
| Linux | The <code>MSG_OOB</code> flag is not set in <code>opts₀</code> . |
| WinXP | The <code>MSG_OOB</code> flag is not set in <code>opts₀</code> and any pending errors are ignored. |

send_10 **udp: block** Block waiting to enqueue datagram

h_0

$\overline{tid \cdot \text{send}(fd, addr, \text{implode } str, opts_0)}$

$h \langle ts :=$
 $ts \oplus (tid \mapsto \text{TIMED}(\text{SEND2}(sid, \uparrow(addr, sock.is_1, \uparrow p'_1, sock.is_2, sock.ps_2),$
 $str, opts),$
 $\text{never_timer})) \rangle;$
 $socks := socks \oplus$
 $\langle [sid, sock \langle es := es; ps_1 := \uparrow p'_1; pr := \text{UDP_PROTO}(udp) \rangle \rangle];$
 $bound := bound;$
 $oq := oq' \rangle$

$h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d);$
 $socks := socks \oplus$
 $\langle [sid, sock \langle es := es; pr := \text{UDP_PROTO}(udp) \rangle \rangle \rangle \wedge$
 $fd \in \text{dom}(h_0.fds) \wedge$
 $fid = h_0.fds[fd] \wedge$
 $h_0.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock.cantsndmore = \mathbf{F} \wedge$
 $(\neg(\text{windows_arch } h.arch) \implies es = *) \wedge$
 $opts = \text{list_to_set } opts_0 \wedge$
 $\neg((\neg \text{bsd_arch } h.arch \wedge \text{MSG_DONTWAIT} \in opts) \vee ff.b(\text{O_NONBLOCK})) \wedge$
 $((\text{linux_arch } h.arch \vee \text{windows_arch } h.arch) \implies \text{MSG_OOB} \notin opts) \wedge$
 $p'_1 \in \text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks) \wedge$
 $(\text{if } sock.ps_1 = * \text{ then } bound = sid :: h_0.bound \text{ else } bound = h_0.bound) \wedge$
 $\text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, \uparrow p'_1, sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F}) \wedge$
 $((addr \neq *) \vee (sock.is_2 \neq *))$

Description

Consider a UDP socket `sid` referenced by `fd` that is not shutdown for writing and has no pending errors. A `send(fd, addr, implode str, opts0)` call is made from thread `tid` which is in the RUN state.

Either the socket is a blocking one: its `O_NONBLOCK` flag is not set, or the call is a blocking one: the `MSG_DONTWAIT` flag is not set in `opts0`.

The socket is either bound to local port `p'_1` or can be autobound to a port `p'_1`. Either the socket has its peer IP address set, or the destination address of the `send()` call is set: `addr` \neq `*`.

A UDP datagram, constructed from the socket's binding quad `sock.is1, $\uparrow p'_1$, sock.is2, sock.ps2`, the destination address argument `addr`, and the data `str`, cannot be placed on the outqueue of the host `oq`.

The call blocks, waiting for the datagram to be enqueued on the host's outqueue. The thread is left in state `SEND2(sid, $\uparrow(addr, sock.is_1, \uparrow p'_1, sock.is_2, sock.ps_2), str, opts)$` . If the socket was autobound to a port then `sid` is appended to the head of the host's list of bound sockets.

Model details

The data to be sent is of type `string` in the `send()` call but is a `byte` list when the datagram is constructed. Here the data, `str` is of type `byte` list and in the transition `implode str` is used to convert it into a string.

The $opts_0$ argument is of type `list`. In the model it is converted to a `set opts` using `list_to_set`. The presence of `MSG_PEEK` is checked for in `opts` rather than in `opts_0`.

Variations

| | |
|---------|---|
| FreeBSD | The <code>MSG_DONTWAIT</code> flag may be set in $opts_0$: it is ignored by FreeBSD. |
| Linux | The <code>MSG_OOB</code> flag must not be set in $opts_0$. |
| WinXP | The <code>MSG_OOB</code> flag must not be set in $opts_0$, and any pending error on the socket is ignored. |

send_11 **udp: fast fail Fail with EAGAIN: call would block and non-blocking behaviour has been requested**

$$\begin{array}{l} \xrightarrow{h_0 \text{ tid.send}(fd, addr, \mathbf{implode} \text{ str}, opts_0)} \\ h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched_timer}}); \\ socks := socks \oplus \\ [(sid, sock \langle es := es; ps_1 := \uparrow p'_1; pr := \text{UDP_PROTO}(udp))] \rangle]; \\ bound := bound; \\ oq := oq' \rangle \end{array}$$

$$\begin{array}{l} h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ socks := socks \oplus \\ [(sid, sock \langle es := es; pr := \text{UDP_PROTO}(udp))] \rangle] \wedge \\ fd \in \mathbf{dom}(h_0.fds) \wedge \\ fid = h_0.fds[fd] \wedge \\ h_0.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ sock.cantsndmore = \mathbf{F} \wedge \\ (\neg(\text{windows_arch } h.arch) \implies es = *) \wedge \\ p'_1 \in \text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks) \wedge \\ (\mathbf{if } sock.ps_1 = * \mathbf{then } bound = sid :: h_0.bound \mathbf{else } bound = h_0.bound) \wedge \\ ((addr \neq *) \vee (sock.is_2 \neq *)) \wedge \\ opts = \mathbf{list_to_set} \text{ } opts_0 \wedge \\ ((\neg \text{bsd_arch } h.arch \wedge \text{MSG_DONTWAIT} \in opts) \vee ff.b(\text{O_NONBLOCK})) \wedge \\ \text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F}) \end{array}$$

Description

Consider a UDP socket sid referenced by fd that is not shutdown for writing and has no pending errors. The thread tid is in the `RUN` state and a call `send(fd, addr, implode str, opts_0)` is made.

The socket is either locally bound to a port p'_1 or can be autobound to a port p'_1 . Either the socket has a peer IP address set, or a destination address was provided in the `send()` call: $addr \neq *$.

Either the socket is non-blocking: its `O_NONBLOCK` flag is set, or the call is non-blocking: `MSG_DONTWAIT` flag was set in the $opts_0$ argument of `send()`.

A UDP datagram (constructed from the socket's binding quad $(sock.is_1, sock.ps_1, sock.is_2, sock.ps_2)$, the destination address argument $addr$, and the data str) cannot be placed on the outqueue of the host oq .

The `send()` call fails with an `EAGAIN` error. A `tid.send(fd, addr, implode str, opts_0)` transition is made, leaving the thread state `FAIL (EAGAIN)`, and the host with outqueue oq' . If the socket was autobound to a port, sid is appended to the host's list of bound sockets.

Model details

The data to be sent is of type `string` in the `send()` call but is a `byte list` when the datagram is constructed. Here the data, str is of type `byte list` and in the transition `implode str` is used to convert it into a string.

The $opts_0$ argument is of type `list`. In the model it is converted to a `set opts` using `list_to_set`. The presence of `MSG_PEEK` is checked for in `opts` rather than in `opts_0`.

Note that on Linux EWOULDBLOCK and EAGAIN are aliased.

Variations

| | |
|---------|--|
| FreeBSD | The socket's O_NONBLOCK flag must be set for the rule to apply; the MSG_DONTWAIT flag is ignored by FreeBSD. |
| WinXP | Pending errors on the socket are ignored. |

send_12 udp: fast fail Fail with ENOTCONN: no peer address set in socket and no destination address provided

$$\begin{aligned}
 & h_0 \\
 & \xrightarrow{tid \cdot \text{send}(fd, *, \text{implode } str, opts_0)} \\
 & h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}); \\
 & \quad socks := socks \oplus \\
 & \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps'_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))]; \\
 & \quad bound := bound \rangle \\
 \\
 & h_0 = h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
 & \quad socks := socks \oplus \\
 & \quad \quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \rangle \wedge \\
 & fd \in \text{dom}(h.fds) \wedge \\
 & fid = h.fds[fd] \wedge \\
 & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 & (\text{if } \text{bsd_arch } h.arch \text{ then } err = \text{EDESTADDRREQ} \\
 & \quad \text{else } err = \text{ENOTCONN}) \wedge \\
 & (\neg(\text{windows_arch } h.arch) \implies es = *) \wedge \\
 & (\text{if } \text{linux_arch } h.arch \text{ then} \\
 & \quad \exists p'_1.p'_1 \in \text{autobind}(ps_1, \text{PROTO_UDP}, h_0.socks) \wedge ps'_1 = \uparrow p'_1 \wedge \\
 & \quad \quad (\text{if } ps_1 = * \text{ then } bound = sid :: h_0.bound \text{ else } bound = h_0.bound) \\
 & \quad \text{else } bound = h_0.bound \wedge ps'_1 = ps_1)
 \end{aligned}$$

Description

Consider a UDP socket *sid* referenced by *fd* that has no pending errors.

A call `send(fd, addr, implode str, opts0)` is made from thread *tid* which is in the RUN state. The socket is either locally bound to a port *p'₁* or it can be autobound to a port *p'₁*.

The socket does not have a peer address set, and no destination address is specified in the `send()` call: *addr* = *. The call will fail with an ENOTCONN error.

A `tid.send(fd, *, implode str, opts0)` transition will be made, leaving the thread in state RET(FAIL ENOTCONN). If the socket was autobound then *sid* is appended to the head of the host's list of bound sockets, *h₀.bound*, resulting in the new list *bound*.

Model details

The data to be sent is of type `string` in the `send()` call but is a `byte` list when the datagram is constructed. Here the data, *str* is of type `byte` list and in the transition `implode str` is used to convert it into a string.

Variations

| | |
|---------|---|
| FreeBSD | On FreeBSD the error returned is EDESTADDRREQ, the socket must not be shut down for writing, and if it is not bound to a local port it will not be autobound. |
| WinXP | Any pending error on the socket is ignored, and if the socket's local port is not bound, <i>ps₁</i> = *, then it will not be autobound. |

send_13 **udp: fast fail** Fail with EMSGSIZE: string to be sent is bigger than UDPpayloadMax

$$h_0 \xrightarrow{tid \cdot \text{send}(fd, addr, \text{implode } str, opts_0)} h \langle [ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EMSGSIZE}))_{\text{sched_timer}}); socks := socks \oplus [(sid, sock \langle [ps_1 := ps'_1; pr := \text{UDP_PROTO}(udp) \rangle]); bound := bound] \rangle;$$

$$h_0 = h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, sock \langle [pr := \text{UDP_PROTO}(udp) \rangle])]] \rangle \wedge$$

$$fd \in \mathbf{dom}(h_0.fds) \wedge$$

$$fid = h_0.fds[fd] \wedge$$

$$h_0.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$(\text{STRLEN}(\text{implode } str) > \text{UDPpayloadMax } h_0.arch \vee$$

$$(\text{bsd_arch } h.arch \wedge \text{STRLEN}(\text{implode } str) > (h_0.socks[sid]).sf.n(\text{SO_SNDBUF}))) \wedge$$

$$ps'_1 \in \{sock.ps_1\} \cup (\mathbf{image}(\uparrow)(\text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks))) \wedge$$

$$(\mathbf{if } sock.ps_1 = * \wedge ps'_1 \neq * \mathbf{then } bound = sid :: h_0.bound \mathbf{else } bound = h_0.bound)$$

Description

Consider a UDP socket *sid* referenced by *fd*. A call `send(fd, addr, implode str, opts0)` is made from thread *tid* which is in the RUN state.

The length in bytes of *str* is greater than `UDPpayloadMax`, the architecture-dependent maximum payload size for a UDP datagram. The `send()` call fails with an `EMSGSIZE` error.

A `tid · send(fd, addr, implode str, opts0)` transition is made leaving the thread in state `RET(FAIL EMSGSIZE)`. Additionally, the socket's local port *ps*₁ may be autobound if it was not bound to a local port when the `send()` call was made. If the autobinding occurs, then the socket's *sid* is added to the list of bound sockets *h*₀.*bound*, leaving the host's list of bound sockets as *bound*.

Model details

The data to be sent is of type `string` in the `send()` call but is a `byte` list when the datagram is constructed. Here the data, *str* is of type `byte` list and in the transition `implode str` is used to convert it into a string.

Variations

| | |
|---------|--|
| FreeBSD | On FreeBSD, the <code>send()</code> call may also fail with <code>EMSGSIZE</code> if the size of <i>str</i> is greater than the value of the socket's <code>SO_SNDBUF</code> option. |
|---------|--|

send_14 **udp: fast fail** Fail with EAGAIN, EADDRNOTAVAIL or ENOBUFS: there are no ephemeral ports left

$$h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \rangle;$$

$$tid \cdot \text{send}(fd, addr, \text{implode } str, opts_0) \rightarrow$$

$$h \langle [ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}); socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, *, es, \text{cantsndmore}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))] \rangle;$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

$$fid = h.fds[fd] \wedge$$

$$h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$\text{cantsndmore} = \mathbf{F} \wedge$$

$$\begin{aligned} &(\neg(\text{windows_arch } h.\text{arch}) \implies es = *) \wedge \\ &\text{autobind}(*, \text{PROTO_UDP}, h.\text{socks}) = \emptyset \wedge \\ &e \in \{\text{EAGAIN}; \text{EADDRNOTAVAIL}; \text{ENOBUFS}\} \end{aligned}$$

Description

Consider a UDP socket sid referenced by fd that is not shutdown for writing and has no pending errors. The socket has no peer address set, and is not bound to a local IP address or port.

From the RUN state, thread tid makes a $\text{send}(fd, addr, \text{implode } str, opts_0)$ call. The socket cannot be auto-bound to an ephemeral port so the call fails. The error returned will be EAGAIN, EADDRNOTAVAIL, or ENOBUFS.

A $tid\text{-send}(fd, addr, \text{implode } str, opts_0)$ transition will be made. The thread will be left in state $RET(\text{FAIL } e)$ where e is one of the above errors.

Model details

The data to be sent is of type `string` in the $\text{send}()$ call but is a `byte list` when the datagram is constructed. Here the data, str is of type `byte list` and in the transition $\text{implode } str$ is used to convert it into a string.

Variations

| | |
|-------|---|
| WinXP | Any pending error on the socket is ignored. |
|-------|---|

send_15 **udp: slow urgent succeed** Return from blocked state after datagram enqueued

$$\begin{aligned} &h \langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d); \\ &socks := socks \oplus \\ &\quad [(sid, sock \langle es := es; pr := \text{UDP_PROTO}(udp) \rangle)] \rangle \\ \xrightarrow{\tau} &h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{""}))_{\text{sched_timer}}); \\ &socks := socks \oplus \\ &\quad [(sid, sock \langle es := es; pr := \text{UDP_PROTO}(udp) \rangle)]; \\ &oq := oq' \rangle \end{aligned}$$

$$\begin{aligned} &sock.\text{cantsndmore} = \mathbf{F} \wedge \\ &(\neg(\text{windows_arch } h.\text{arch}) \implies es = *) \wedge \\ &\text{STRLEN}(\text{implode } str) \leq \text{UDPpayloadMax } h.\text{arch} \wedge \\ &(\text{dosend}(h.\text{ifds}, h.\text{rttab}, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T}) \vee \\ &\quad \text{dosend}(h.\text{ifds}, h.\text{rttab}, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h.oq, oq', \mathbf{T})) \wedge \\ &(addr \neq * \vee sock.is_2 \neq * \vee is_2 \neq *) \end{aligned}$$

Description

Consider a UDP socket sid that is not shutdown for writing and has no pending errors. The thread tid is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

A datagram can be constructed using str as its data. The length in bytes of str is less than or equal to UDPpayloadMax , the architecture-dependent maximum payload size for a UDP datagram. There are three possible destination addresses:

- $addr$, the destination address specified in the $\text{send}()$ call.
- is_2, ps_2 , the socket's peer address when the $\text{send}()$ call was made.
- $sock.is_2, sock.ps_2$, the socket's current peer address.

At least one of $addr$, is_2 , and $sock.is_2$ must specify an IP address: they are not all set to $*$. One of the three addresses will be used as the destination address of the datagram. The datagram can be successfully enqueued on the host's outqueue, $h.oq$, resulting in a new outqueue oq' .

An τ transition is made, leaving the thread state $\text{RET}(\text{OK}(\text{""}))$, and the host with new outqueue oq' .

send_16 udp: slow urgent fail Fail: blocked socket has entered an error state

$$\begin{aligned}
 & h \{ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str))_d); \\
 & socks := socks \oplus \\
 & \quad [(sid, sock \{es := \uparrow e; pr := \text{UDP_PROTO}(udp)\})]\} \\
 \xrightarrow{\tau} & h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}); \\
 & socks := socks \oplus \\
 & \quad [(sid, sock \{es := *; pr := \text{UDP_PROTO}(udp)\})]\}
 \end{aligned}$$

$\neg(\text{windows_arch } h.\text{arch})$

Description

Consider a UDP socket sid that has pending error $\uparrow e$. The thread tid is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$. The error, e , will be returned to the caller.

At τ transition is made, leaving the thread state $\text{RET}(\text{FAIL } e)$.

Note that the error has occurred after the thread entered the SEND2 state: rule *send_11* specifies that the call cannot block if there is a pending error.

Variations

| | |
|-------|--|
| WinXP | This rule does not apply: all pending errors on a socket are ignored for a <code>send()</code> call. |
|-------|--|

send_17 udp: slow urgent fail Fail with **EMSGSIZE** or **ENOTCONN**: blocked socket has had peer address unset or string to be sent is too big

$$\begin{aligned}
 & h \{ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d); \\
 & socks := socks \oplus \\
 & \quad [(sid, sock \{sf := sf; es := es; pr := \text{UDP_PROTO}(udp)\})]\} \\
 \xrightarrow{\tau} & h \{ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}); \\
 & socks := socks \oplus \\
 & \quad [(sid, sock \{sf := sf; es := es; pr := \text{UDP_PROTO}(udp)\})]\}
 \end{aligned}$$

$(\neg(\text{windows_arch } h.\text{arch}) \implies es = *) \wedge$
 $(\exists oq'. \text{dosend}(h.\text{ifds}, h.\text{rttab}, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T})) \wedge$
 $((\text{STRLEN}(\mathbf{implode } str) > \text{UDPpayloadMax } h.\text{arch} \wedge (e = \text{EMSGSIZE})) \vee$
 $(\text{bsd_arch } h.\text{arch} \wedge \text{STRLEN}(\mathbf{implode } str) > sf.n(\text{SO_SNDBUF}) \wedge (e = \text{EMSGSIZE})) \vee$
 $((sock.is_2 = *) \wedge (addr = *) \wedge (e = \text{ENOTCONN})))$

Description

Consider a UDP socket sid with no pending errors. The thread tid is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

A datagram is constructed with str as its payload. Its destination address is taken from $addr$, the destination address specified when the `send()` call was made, or (is_2, ps_2) , the socket's peer address when the `send()` call was made. It is possible to enqueue the datagram on the host's outqueue, $h.oq$.

This rule covers two cases. In the first, the length in bytes of str is greater than `UDPpayloadMax`, the architecture-dependent maximum payload size for a UDP datagram. The error `EMSGSIZE` is returned.

In the second case, the original `send()` call did not have a destination address specified: $addr = *$, and the socket has had the IP address of its peer address unset: $sock.is_2 = *$. The peer address of the socket when the `send()` call was made, (is_2, ps_2) , is ignored, and an `ENOTCONN` error is returned.

In either case, a τ transition is made, leaving the thread state $\text{RET}(\text{FAIL } e)$ where e is either EMSGSIZE or ENOTCONN .

Variations

| | |
|---------|--|
| FreeBSD | An EMSGSIZE error can also be returned if the size of str is greater than the value of the socket's SO_SNDBUF option. |
| WinXP | Any pending error on the socket is ignored. |

send_18 udp: fast fail Fail with EOPNOTSUPP: MSG_PEEK flag not supported for send() calls on WinXP; or MSG_OOB flag not supported on WinXP and Linux

$$h_0 \xrightarrow{\text{tid.send}(fd, addr, \text{implode } str, \text{opts}_0)} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } \text{EOPNOTSUPP}))_{\text{sched_timer}}); \text{socks} := \text{socks} \oplus [(sid, sock \langle \langle ps_1 := ps'_1; pr := \text{UDP_PROTO}(udp) \rangle \rangle)]; \text{bound} := \text{bound} \rangle \rangle;$$

$$h_0 = h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{socks} := \text{socks} \oplus [(sid, sock \langle \langle ps_1 := ps_1; pr := \text{UDP_PROTO}(udp) \rangle \rangle)] \rangle \rangle \wedge$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

$$fid = h.fds[fd] \wedge$$

$$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$\text{opts} = \mathbf{list_to_set } \text{opts}_0 \wedge$$

$$((\text{MSG_PEEK} \in \text{opts} \wedge \text{windows_arch } h.arch) \vee$$

$$(\text{MSG_OOB} \in \text{opts} \wedge \text{sock.cantsndmore} = \mathbf{F} \wedge (\text{linux_arch } h.arch \vee \text{windows_arch } h.arch))) \wedge$$

$$\mathbf{(if } \text{linux_arch } h.arch \mathbf{ then}$$

$$\quad \exists p'_1.p'_1 \in \text{autobind}(ps_1, \text{PROTO_UDP}, h_0.\text{socks}) \wedge ps'_1 = \uparrow p'_1 \wedge$$

$$\quad \mathbf{(if } ps_1 = * \mathbf{ then } \text{bound} = sid :: h_0.\text{bound} \mathbf{ else } \text{bound} = h_0.\text{bound})}$$

$$\mathbf{ else}$$

$$\quad ps_1 = ps'_1 \wedge \text{bound} = h_0.\text{bound})$$

Description

Consider a UDP socket sid referenced by fd . From thread tid , which is in the RUN state, a $\text{send}(fd, addr, \text{implode } str, \text{opts}_0)$ call is made.

This rule covers two cases. In the first, on WinXP, the MSG_PEEK flag is set in opts_0 . In the second case, on Linux and WinXP, the socket has not been shut down for writing, and the MSG_OOB flag is set in opts_0 . In either case, the $\text{send}()$ call fail with an EOPNOTSUPP error.

A $\text{tid.send}(fd, addr, \text{implode } str, \text{opts}_0)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL } \text{EOPNOTSUPP})$.

Model details

The opts_0 argument is of type list . In the model it is converted to a $\text{set } \text{opts}$ using $\mathbf{list_to_set}$. The presence of MSG_PEEK is checked for in opts rather than in opts_0 .

Variations

| | |
|---------|--|
| FreeBSD | FreeBSD ignores the MSG_PEEK and MSG_OOB flags for $\text{send}()$. |
| Linux | Linux ignores the MSG_PEEK flag for $\text{send}()$. |

send_19 **udp: fast fail** Fail with EADDRINUSE: on FreeBSD, local and destination address quad in use by another socket

$$\frac{h_0}{tid \cdot \text{send}(fd, \uparrow(i_2, p_2), \text{implode } str, opts_0)} \quad h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EADDRINUSE}))_{\text{sched_timer}}); \rangle \rangle$$

$$\begin{aligned} & socks := socks \oplus \\ & \quad [(sid, sock)]; \\ & bound := bound \end{aligned}$$

$$\begin{aligned} & \text{bsd_arch } h.arch \wedge \\ & h_0 = h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \rangle \rangle \\ & \quad socks := socks \oplus \\ & \quad \quad [(sid, sock)] \rangle \rangle \wedge \\ & \text{sock.cantsndmore} = \mathbf{F} \wedge \\ & (\neg(\text{windows_arch } h.arch) \implies \text{sock.es} = *) \wedge \\ & p'_1 \in \text{autobind}(\text{sock.ps}_1, \text{PROTO_UDP}, h_0.socks) \wedge \\ & (\text{if } \text{sock.ps}_1 = * \text{ then } bound = sid :: h_0.bound \text{ else } bound = h_0.bound) \wedge \\ & i'_1 \in \text{auto_outroute}(i_2, \text{sock.is}_1, h_0.rttab, h_0.ifds) \wedge \\ & fd \in \text{dom}(h_0.fds) \wedge \\ & fid = h_0.fds[fd] \wedge \\ & h_0.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ & \text{sock} = (h_0.socks[sid]) \wedge \\ & \text{proto_of } \text{sock.pr} = \text{PROTO_UDP} \wedge \\ & (\exists sid'. \\ & \quad sid' \in \text{dom}(h_0.socks) \wedge \\ & \quad \text{let } s = h_0.socks[sid'] \text{ in} \\ & \quad s.is_1 = \uparrow i'_1 \wedge s.ps_1 = \uparrow p'_1 \wedge \\ & \quad s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge \\ & \quad \text{proto_of } s.pr = \text{PROTO_UDP}) \end{aligned}$$

Description

On FreeBSD, consider a UDP socket sid referenced by fd that is not shutdown for writing. From thread tid , which is in the RUN state, a $\text{send}(fd, \uparrow(i_2, p_2), \text{implode } str, opts_0)$ call is made. The socket is bound to local port p'_1 or it can be autobound to port p'_1 . The socket can be bound to a local IP address i'_1 which has a route to i_2 . Another socket, sid' , is locally bound to (i'_1, p'_1) and has its peer address set to (i_2, p_2) . The $\text{send}()$ call will fail with an EADDRINUSE error.

A $tid \cdot \text{send}(fd, \uparrow(i_2, p_2), \text{implode } str, opts_0)$ transition will be made, leaving the thread state $\text{RET}(\text{FAIL EADDRINUSE})$.

Variations

| | |
|-------|---------------------------|
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

send_21 **udp: fast fail** Fail with EISCONN: socket has peer address set and destination address is specified in call on FreeBSD

$$\frac{h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \rangle \rangle \quad socks := socks \oplus \quad [(sid, sock \langle \langle es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP_PROTO}(udp) \rangle \rangle)] \rangle \rangle}{tid \cdot \text{send}(fd, \uparrow(i_3, p_3), \text{implode } str, opts_0)}$$

$$\begin{aligned}
&h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EISCONN}))_{\text{sched_timer}}); \\
&\text{socks} := \text{socks} \oplus \\
&\quad [(sid, \text{sock} \langle es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP_PROTO}(udp) \rangle \rangle)]
\end{aligned}$$

$$\begin{aligned}
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&\text{bsd_arch } h.arch
\end{aligned}$$

Description

Consider a UDP socket sid referenced by fd that has its peer address set: $is_2 = \uparrow i_2$, and $ps_2 = \uparrow p_2$. From thread tid , which is in the RUN state, a $\text{send}(fd, \uparrow(i_3, p_3), \mathbf{implode} \text{ } str, \text{opts}_0)$ call is made. On FreeBSD, the call will fail with the EISCONN error, as the call specified a destination address even though the socket has a peer address set.

A $tid.\text{send}(fd, \uparrow(i_3, p_3), \mathbf{implode} \text{ } str, \text{opts}_0)$ transition will be made, leaving the thread state $\text{RET}(\text{FAIL EISCONN})$.

Variations

| | |
|-------|---|
| Posix | If the socket is connectionless-mode, the message shall be sent to the address specified by $\uparrow(i_3, p_3)$. See the above $\text{send}()$ rules. |
| Linux | This rule does not apply. Linux allows the $\text{send}()$ call to occur. See the above $\text{send}()$ rules. |
| WinXP | This rule does not apply. WinXP allows the $\text{send}()$ call to occur. See the above $\text{send}()$ rules. |

send_22 **udp: fast fail** Fail with EPIPE or ESHUTDOWN: socket shut down for writing

$$\begin{aligned}
&h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
&\text{socks} := \text{socks} \oplus \\
&\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))]
\end{aligned}$$

$$\text{tid.send}(fd, \text{addr}, \mathbf{implode} \text{ } str, \text{opts}_0)$$

$$\begin{aligned}
&h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched_timer}}); \\
&\text{socks} := \text{socks} \oplus \\
&\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, \text{cantrcvmore}, \text{UDP_PROTO}(udp)))]
\end{aligned}$$

$$\begin{aligned}
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&\mathbf{if} \text{ windows_arch } h.arch \mathbf{then} \text{ err} = \text{ESHUTDOWN} \\
&\mathbf{else} \text{ err} = \text{EPIPE}
\end{aligned}$$

Description

From thread tid , which is in the RUN state, a $\text{send}(fd, \text{addr}, \mathbf{implode} \text{ } str, \text{opts}_0)$ call is made where fd refers to a UDP socket sid that is shut down for writing. The call fails with an EPIPE error.

A $tid.\text{send}(fd, \text{addr}, \mathbf{implode} \text{ } str, \text{opts}_0)$ transition is made, leaving the thread in state $\text{RET}(\text{FAIL EPIPE})$.

Variations

| | |
|-------|---|
| WinXP | The call fails with an ESHUTDOWN error rather than EPIPE. |
|-------|---|

| | |
|---|--|
| <p><i>send_23</i> udp: fast fail Fail with pending error</p> $ \begin{array}{l} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ socks := socks \oplus \\ \quad [(sid, sock \langle es := \uparrow e \rangle)] \rangle \\ \hline tid \cdot \text{send}(fd, addr, \mathbf{implode} \ str, opts_0) \rightarrow \end{array} $ <p> $h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}});$ $socks := socks \oplus$ $\quad [(sid, sock \langle es := * \rangle)] \rangle$ </p> <p> $fd \in \mathbf{dom}(h.fds) \wedge$ $fd = h.fds[fd] \wedge$ $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$ $\text{proto_of } sock.pr = \text{PROTO_UDP} \wedge$ $\neg(\text{windows_arch } h.arch)$ </p> | |
|---|--|

Description

From thread *tid*, which is in the RUN state, a `send(fd, addr, implode str, opts0)` call is made where *fd* refers to a UDP socket *sid* that has pending error $\uparrow e$. The call fails, returning the pending error.

A `tid·send(fd, addr, implode str, opts0)` transition is made, leaving the thread in state `RET(FAIL e)`.

Variations

| | |
|-------|--|
| WinXP | This rule does not apply: all pending errors are ignored for <code>send()</code> calls on WinXP. |
|-------|--|

15.23 setfileflags() (TCP and UDP)

`setfileflags` : (fd * filebflag list) → unit

A call to `setfileflags(fd, flags)` sets the flags on a file referred to by *fd*. *flags* is the list of file flags to set. The possible flags are:

- `O_ASYNC` Specifies whether signal driven I/O is enabled.
- `O_NONBLOCK` Specifies whether a socket is non-blocking.

The call returns successfully if the flags were set, or fails with an error otherwise.

15.23.1 Errors

A call to `setfileflags()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------|--|
| EBADF | The file descriptor passed is not a valid file descriptor. |
|-------|--|

15.23.2 Common cases

setfileflags_1; *return_1*

15.23.3 API

`setfileflags()` is Posix `fcntl(fd,F_GETFL,flags)`. On WinXP it is `ioctlsocket()` with the `FIONBIO` command.

```
Posix:    int fcntl(int fildes, int cmd, ...);
FreeBSD:  int fcntl(int fd, int cmd, ...);
Linux:    int fcntl(int fd, int cmd);
WinXP:    int ioctlsocket(SOCKET s, long cmd, u_long* argp)
```

In the Posix interface:

- `fildes` is a file descriptor for the file to retrieve flags from. It corresponds to the `fd` argument of the model `setfileflags()`. On WinXP the `s` is a socket descriptor corresponding to the `fd` argument of the model `setfileflags()`.
- `cmd` is a command to perform an operation on the file. This is set to `F_GETFL` for the model `setfileflags()`. On WinXP, `cmd` is set to `FIONBIO` to get the `O_NONBLOCK` flag; there is no `O_ASYNC` flag on WinXP.
- The call takes a variable number of arguments. For the model `setfileflags()` it takes three arguments: the two described above and a third of type `long` which represents the list of flags to set, corresponding to the `flags` argument of the model `setfileflags()`. On WinXP this is the `argp` argument.
- The returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.23.4 Model details

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `WSAENOTSOCK` is a possible error on WinXP as the `ioctlsocket()` call is specific to a socket. In the model the `setfileflags()` call is performed on a file.

15.23.5 Summary

setfileflags_1 **all: fast succeed** Update all the file flags for an open file description

15.23.6 Rules

setfileflags_1 **all: fast succeed** Update all the file flags for an open file description

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{files} := \text{files} \oplus [(fid, \text{FILE}(ft, ff \langle b := ffb \rangle)] \rangle}{tid \cdot \text{setfileflags}(fd, flags) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \text{files} := \text{files} \oplus [(fid, \text{FILE}(ft, ff \langle b := ffb' \rangle)] \rangle}$$

$$\begin{aligned} fd &\in \mathbf{dom}(h.fds) \wedge \\ fd &= h.fds[fd] \wedge \\ ffb' &= \lambda x.x \in flags \end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a `setfileflags(fd, flags)` call is made. *fd* refers to the open file description (*fid*, `FILE(ft, ff (b := ffb))`) where *ffb* is the set of boolean file flags currently set. *flags* is a list of boolean file flags, possibly containing duplicates.

All of the boolean file flags for the file description will be updated. The flags in *flags* will all be set to **T**, and all other flags will be set to **F**, resulting in a new set of boolean file flags, *ffb'*.

A `tid.setfileflags(fd, flags)` transition is made, leaving the thread state `RET(OK())`.

Note this is not exactly the same as `getfileflags_1`: `getfileflags` never returns duplicates, but duplicates may be passed to `setfileflags`.

15.24 setsockbopt() (TCP and UDP)

`setsockbopt : (fd * sockbflag * bool) → unit`

A call `setsockbopt(fd, f, b)` sets the value of one of a socket's boolean flags.

Here the *fd* argument is a file descriptor referring to a socket on which to set a flag, *f* is the boolean socket flag to set, and *b* is the value to set it to. Possible boolean flags are:

- `SO_BSDCOMPAT` Specifies whether the BSD semantics for delivery of ICMPs to UDP sockets with no peer address set is enabled.
- `SO_DONTROUTE` Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address.
- `SO_KEEPALIVE` Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol.
- `SO_OOBINLINE` Leaves received out-of-band data (data marked urgent) inline.
- `SO_REUSEADDR` Specifies that the rules used in validating addresses supplied to `bind()` should allow reuse of local ports, if this is supported by the protocol.

15.24.1 Errors

A call to `setsockbopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------------|--|
| ENOPROTOOPT | The option is not supported by the protocol. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.24.2 Common cases

`setsockbopt_1`; `return_1`

15.24.3 API

`setsockbopt()` is Posix `setsockopt()` for boolean-valued socket flags.


```

Posix:      int setsockopt(int socket, int level, int option_name,
                    const void *option_value,
                    socklen_t option_len);
FreeBSD:    int setsockopt(int s, int level, int optname,
                    const void *optval, socklen_t optlen);
Linux:      int setsockopt(int s, int level, int optname,
                    const void *optval, socklen_t optlen);
WinXP:      int setsockopt(SOCKET s, int level, int optname,
                    const char* optval,int optlen);

```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the `fd` argument of the model `setsockopt()`.
- `level` is the protocol level at which the flag resides: `SOL_SOCKET` for the socket level options, and `option_name` is the flag to be set. These two correspond to the `flag` argument of the model `setsockopt()` where the possible values of `option_name` are limited to: `SO_BSDCOMPAT`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_OOINLINE`, and `SO_REUSEADDR`.
- `option_value` is a pointer to a location of size `option_len` containing the value to set the flag to. These two correspond to the `b` argument of type `bool` in the model `setsockopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.24.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `setsockopt()`.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.24.5 Summary

| | | |
|---------------------|--------------------------|---|
| <i>setsockopt_1</i> | all: fast succeed | Successfully set a boolean socket flag |
| <i>setsockopt_2</i> | udp: fast fail | Fail with <code>ENOPROTOOPT</code> : <code>SO_KEEPALIVE</code> and <code>SO_OOINLINE</code> options not supported for a UDP socket on WinXP |

15.24.6 Rules

setsockopt_1 **all: fast succeed** Successfully set a boolean socket flag

$$\begin{array}{c}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_a); \\
 socks := socks \oplus [(sid, sock)] \rangle
 \end{array}
 \xrightarrow{tid \cdot \text{setsockopt}(fd, f, b)}
 \begin{array}{c}
 h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \\
 socks := socks \oplus [(sid, sock')] \rangle
 \end{array}$$

$$\begin{array}{l}
 fd \in \mathbf{dom}(h.fds) \wedge \\
 fid = h.fds[fd] \wedge \\
 h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge
 \end{array}$$

$$sock' = sock \langle [sf := sock.sf \langle [b := sock.sf.b \oplus (f \mapsto b)]] \rangle$$

$$\wedge$$

$$(windows_arch \ h.arch \wedge \ proto_of \ sock.pr = \text{PROTO_UDP}$$

$$\implies f \notin \{SO_KEEPALIVE; SO_OOBINLINE\})$$

Description

Consider a socket sid , referenced by fd , and with socket flags $sock.sf$. From thread tid , which is in the RUN state, a $setsockopt(fd, f, b)$ call is made. f is the boolean socket flag to be set, and b is the boolean value to set it to. The call succeeds.

A $tid.setsockopt(fd, f, b)$ is made, leaving the thread state $\text{RET}(\text{OK}())$. The socket's boolean flags, $sock.sf.b$, are updated such that f has the value b .

Variations

| | |
|-------|---|
| WinXP | As above, except that if sid is a UDP socket, then f cannot be $SO_KEEPALIVE$ or $SO_OOBINLINE$. |
|-------|---|

setsockopt_2 **udp: fast fail** Fail with **ENOPROTOOPT: SO_KEEPALIVE and SO_OOBINLINE options not supported for a UDP socket on WinXP**

$$h \langle [ts := ts \oplus (tid \mapsto (\text{RUN})_d)] ;$$

$$socks := socks \oplus$$

$$[(sid, sock \langle [pr := \text{UDP_PROTO}(udp)]]]]]$$

$$\xrightarrow{tid.setsockopt(fd, f, b)} h \langle [ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}) ;$$

$$socks := socks \oplus$$

$$[(sid, sock \langle [pr := \text{UDP_PROTO}(udp)]]]]]$$

$$windows_arch \ h.arch \wedge$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$

$$fd = h.fds[fd] \wedge$$

$$h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$$

$$f \in \{SO_KEEPALIVE; SO_OOBINLINE\}$$

Description

On WinXP, consider a UDP socket sid referenced by fd . From thread tid , which is in the RUN state, a $setsockopt(fd, f, b)$ call is made, where f is either $SO_KEEPALIVE$ or $SO_OOBINLINE$. The call fails with an ENOPROTOOPT error.

A $tid.setsockopt(fd, f, b)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOPROTOOPT})$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

15.25 setsockopt() (TCP and UDP)

$setsockopt : (fd * socknflag * int) \rightarrow \text{unit}$

A call `setsockopt(fd, f, n)` sets the value of one of a socket's numeric flags. The `fd` argument is a file descriptor referring to a socket to set a flag on, `f` is the numeric socket flag to set, and `n` is the value to set it to. Possible numeric flags are:

- `SO_RCVBUF` Specifies the receive buffer size.
- `SO_RCVLOWAT` Specifies the minimum number of bytes to process for socket input operations.
- `SO_SNDBUF` Specifies the send buffer size.
- `SO_SNDLOWAT` Specifies the minimum number of bytes to process for socket output operations.

15.25.1 Errors

A call to `setsockopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|--------------------------|--|
| <code>EINVAL</code> | On FreeBSD, attempting to set a numeric flag to zero. |
| <code>ENOPROTOOPT</code> | The option is not supported by the protocol. |
| <code>EBADF</code> | The file descriptor passed is not a valid file descriptor. |
| <code>ENOTSOCK</code> | The file descriptor passed does not refer to a socket. |

15.25.2 Common cases

setsockopt_1; return_1

15.25.3 API

`setsockopt()` is Posix `setsockopt()` for numeric-valued socket flags.

```
Posix:      int setsockopt(int socket, int level, int option_name,
                        const void *option_value,
                        socklen_t option_len);
FreeBSD:    int setsockopt(int s, int level, int optname,
                        const void *optval, socklen_t optlen);
Linux:      int setsockopt(int s, int level, int optname,
                        const void *optval, socklen_t optlen);
WinXP:      int setsockopt(SOCKET s, int level, int optname,
                        const char* optval,int optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the `fd` argument of the model `setsockopt()`.
- `level` is the protocol level at which the flag resides: `SOL_SOCKET` for the socket level options, and `option_name` is the flag to be set. These two correspond to the *flag* argument of the model `setsockopt()` where the possible values of `option_name` are limited to: `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, and `SO_SNDLOWAT`.
- `option_value` is a pointer to a location of size `option_len` containing the value to set the flag to. These two correspond to the `n` argument of type `int` in the model `setsockopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.25.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.

- EINTR signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `setsockopt()`.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.25.5 Summary

| | | |
|---------------------------|--------------------------|--|
| <code>setsockopt_1</code> | all: fast succeed | Successfully set a numeric socket flag |
| <code>setsockopt_2</code> | all: fast fail | Fail with EINTR: on FreeBSD numeric socket flags cannot be set to zero |
| <code>setsockopt_4</code> | all: fast fail | Fail with ENOPROTOOPT: SO_SNDLOWAT not settable on Linux |

15.25.6 Rules

`setsockopt_1` **all: fast succeed** Successfully set a numeric socket flag

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, sock)] \rangle \xrightarrow{tid \cdot \text{setsockopt}(fd, f, n)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched.timer}}); socks := socks \oplus [(sid, sock')] \rangle$$

$fd \in \text{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $n' = \max(sf_min_n \ h.arch \ f)(\min(sf_max_n \ h.arch \ f)(\text{clip_int_to_num } n)) \wedge$
 $ns = (\text{if } \text{bsd_arch } h.arch \wedge f = \text{SO_SNDBUF} \wedge n' < sock.sf.n(\text{SO_SNDLOWAT}) \text{ then}$
 $\quad (sock.sf.n \oplus (f \mapsto n')) \oplus (\text{SO_SNDLOWAT} \mapsto n')$
 $\quad \text{else } sock.sf.n \oplus (f \mapsto n')) \wedge$
 $sock' = sock \langle sf := sock.sf \langle n := ns \rangle \rangle$

Description

Consider the socket `sid`, referenced by `fd`, with numeric socket flags `sock.sf.n`. From the thread `tid`, which is in the RUN state, a `setsockopt(fd, f, n)` call is made where `f` is a numeric socket flag to be updated, and `n` is the integer value to set it to. The call succeeds.

A `tid.setsockopt(fd, f, n)` transition is made, leaving the thread state `RET(OK())`. The socket's numeric flag `f` is updated to be the value `n'` which is: the architecture-specific minimum value for `f` `sf_min_n h.arch f`, if `n` is less than this value; the architecture-specific maximum value for `f`, i.e. `sf_max_n h.arch f`, if `n` is greater than this value, or `n` otherwise.

Variations

| | |
|---------|---|
| FreeBSD | If the flag to be set is SO_SNDLOWAT and the new value <code>n</code> is less than the value of the socket's SO_SNDLOWAT flag then the SO_SNDLOWAT flag is also set to <code>n</code> . |
|---------|---|

`setsockopt_2` **all: fast fail** Fail with EINTR: on FreeBSD numeric socket flags cannot be set to zero

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle$$

$$\underline{tid \cdot \text{setsockopt}(fd, f, n)} \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}) \rrbracket$$

clip_int_to_num $n = 0 \wedge$
 bsd_arch $h.arch$

Description

On FreeBSD, from thread tid , which is in the RUN state, a $\text{setsockopt}(fd, f, n)$ call is made where fd is a file descriptor, f is a numeric socket flag, and n is an integer value to set f to. Because the numeric value of n equals 0, the call fails with an EINVAL error.

A $tid \cdot \text{setsockopt}(fd, f, n)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EINVAL})$.

Variations

| | |
|-------|---------------------------|
| Posix | This rule does not apply. |
| Linux | This rule does not apply. |
| WinXP | This rule does not apply. |

setsockopt_4 **all: fast fail** Fail with ENOPROTOOPT: SO_SNDLOWAT not settable on Linux

$$\underline{tid \cdot \text{setsockopt}(fd, f, n)} \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket$$

$$\underline{tid \cdot \text{setsockopt}(fd, f, n)} \rightarrow h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}) \rrbracket$$

linux_arch $h.arch \wedge$
 $f = \text{SO_SNDLOWAT}$

Description

On Linux, from thread tid , which is in the RUN state, a $\text{setsockopt}(fd, f, n)$ call is made. $f = \text{SO_SNDLOWAT}$, which is not settable, so the call fails with an ENOPROTOOPT error.

A $tid \cdot \text{setsockopt}(fd, f, n)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOPROTOOPT})$.

Variations

| | |
|---------|--|
| FreeBSD | This rule does not apply. |
| WinXP | This rule does not apply. Note the warning from the Win32 docs (at MSDN setsockopt): "If the setsockopt function is called before the bind function, TCP/IP options will not be checked with TCP/IP until the bind occurs. In this case, the setsockopt function call will always succeed, but the bind function call may fail because of an early setsockopt failing." This is currently unimplemented. |

15.26 setsockopt() (TCP and UDP)

setsockopt : (fd * socktflag * (int * int) option) → unit

A call `setsockopt(fd, f, t)` sets the value of one of a socket's time-option flags.

The `fd` argument is a file descriptor referring to a socket to set a flag on, `f` is the time-option socket flag to set, and `t` is the value to set it to. Possible time-option flags are:

- `SO_RCVTIMEO` Specifies the timeout value for input operations.
- `SO_SNDTIMEO` Specifies the timeout value that an output function blocks because flow control prevents data from being sent.

If `t = *` then the timeout is disabled. If `t = ↑(s, ns)` then the timeout is set to `s` seconds and `ns` nanoseconds.

15.26.1 Errors

A call to `setsockopt()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-------------|--|
| EBADF | The file descriptor <code>fd</code> does not refer to a valid file descriptor. |
| EDOM | The timeout value is too big to fit in the socket structure. |
| ENOPROTOOPT | The option is not supported by the protocol. |
| ENOTSOCK | The file descriptor <code>fd</code> does not refer to a socket. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |

15.26.2 Common cases

setsockopt_1; return_1

15.26.3 API

`setsockopt()` is Posix `setsockopt()` for time-option socket flags.

```
Posix:      int setsockopt(int socket, int level, int option_name,
                        const void *option_value,
                        socklen_t option_len);
```

```
FreeBSD:   int setsockopt(int s, int level, int optname,
                        const void *optval, socklen_t optlen);
```

```
Linux:     int setsockopt(int s, int level, int optname,
                        const void *optval, socklen_t optlen);
```

```
WinXP:    int setsockopt(SOCKET s, int level, int optname,
                        const char* optval, int optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the `fd` argument of the model `setsockopt()`.
- `level` is the protocol level at which the flag resides: `SOL_SOCKET` for the socket level options, and `option_name` is the flag to be set. These two correspond to the *flag* argument of the model `setsockopt()` where the possible values of `option_name` are limited to: `SO_RCVTIMEO` and `SO_SNDTIMEO`.
- `option_value` is a pointer to a location of size `option_len` containing the value to set the flag to. These two correspond to the `t` argument of type `(int *int) option` in the model `setsockopt()`.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

15.26.4 Model details

The following errors are not modelled:

- EFAULT signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.
- EINVAL signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to `setsockopt()`.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.26.5 Summary

| | | |
|---------------------------|--------------------------|---|
| <code>setsockopt_1</code> | all: fast succeed | Successfully set a time-option socket flag |
| <code>setsockopt_4</code> | all: fast fail | Fail with ENOPROTOOPT: on WinXP SO_LINGER not settable for a UDP socket |
| <code>setsockopt_5</code> | all: fast fail | Fail with EDOM: timeout value too long to fit in socket structure |

15.26.6 Rules

`setsockopt_1` **all: fast succeed** Successfully set a time-option socket flag

$$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, sock)] \rangle \xrightarrow{tid \cdot \text{setsockopt}(fd, f, t)} h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); socks := socks \oplus [(sid, sock')] \rangle$$

$$\begin{aligned} & fd \in \mathbf{dom}(h.fds) \wedge \\ & fid = h.fds[fd] \wedge \\ & h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ & \text{tltimeopt_wf } t \wedge \\ & t' = \text{time_of_tltimeopt } t \wedge \\ & t' \geq 0 \wedge \\ & (\text{if } f \in \{\text{SO_RCVTIMEO}; \text{SO_SNDBTIMEO}\} \wedge t' = 0 \\ & \text{then } t'' = \infty \\ & \text{else } t'' = t') \wedge \\ & (\text{if } f = \text{SO_LINGER} \wedge t = \uparrow(s, ns) \text{ then } ns = 0 \text{ else } \mathbf{T}) \wedge \\ & (f \in \{\text{SO_RCVTIMEO}; \text{SO_SNDBTIMEO}\} \implies t'' = \infty \vee t'' \leq \text{sndrcv_timeo_t_max}) \wedge \\ & sock' = sock \langle sf := sock.sf \langle t := sock.sf.t \oplus (f \mapsto t'') \rangle \rangle \end{aligned}$$

Description

From thread `tid`, which is in the `RUN` state, a `setsockopt(fd, f, t)` call is made. `fd` refers to a socket `sid` which has time-option socket flags `sock.sf.t`; `f` is a time-option socket flag: either `SO_RCVTIMEO` or `SO_SNDTIMEO`; and `t` is the well formed time-option value to set `f` to. The call succeeds.

A `tid.setsockopt(fd, f, t)` transition is made, leaving the thread state `RET(OK())`. If `t = *` or `t = ↑(0, 0)` then the socket's time-option flags are updated such that `sock.sf.t(f) = *`, representing ∞ ; otherwise the socket's time-option flags are updated such that `f` has the time value represented by `t`, which must be less than `snd_rcv_timeo_t_max`.

Model details

The type of `t` is `(int * int) option`, but the type of a time-option socket flag is `time`. The auxiliary function `time_of_tltimeopt` is used to do the conversion.

setsockopt_4 **all: fast fail** Fail with ENOPROTOOPT: on WinXP SO_LINGER not settable for a UDP socket

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{setsockopt}(fd, f, t)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched_timer}}) \rangle$$

windows_arch $h.arch \wedge$
 $fd \in \text{dom}(h.fds) \wedge fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $\text{proto_of}(h.socks[sid]).pr = \text{PROTO_UDP} \wedge$
 $f = \text{SO_LINGER}$

Description

On WinXP, from thread tid , which is in the RUN state, a $\text{setsockopt}(fd, f, t)$ call is made. fd is a file descriptor referring to a UDP socket sid , f is the time-option socket SO_LINGER. The flag f is not settable, so the call fails with an ENOPROTOOPT error.

A $tid \cdot \text{setsockopt}(fd, f, t)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOPROTOOPT})$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
| Linux | This rule does not apply. |

setsockopt_5 **all: fast fail** Fail with EDOM: timeout value too long to fit in socket structure

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle}{tid \cdot \text{setsockopt}(fd, f, t)} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EDOM}))_{\text{sched_timer}}) \rangle$$

$f \in \{\text{SO_RCVTIMEO}; \text{SO_SNDBTIMEO}\} \wedge$
 $\text{tltimeopt_wf } t \wedge$
 $t' = \text{time_of_tltimeopt } t \wedge$
(if $t' = 0$
then $t'' = \infty$
else $t'' = t') \wedge$
 $\neg(t'' = \infty \vee t'' \leq \text{sndrcv_timeo_t_max})$

Description

From thread tid , which is currently in the RUN state, a $\text{setsockopt}(fd, f, t)$ call is made. f is a time-option socket flag that is either SO_RCVTIMEO or SO_SNDTIMEO, and t is the time value to set f to. The call fails with an EDOM error because the value t is too large to fit in the socket structure: it is not zero and it is greater than $\text{sndrcv_timeo_t_max}$.

A $tid \cdot \text{setsockopt}(fd, f, t)$ call is made, leaving the thread state $\text{RET}(\text{FAIL EDOM})$.

Model details

The type of t is $(\text{int} * \text{int})$ option, but the type of a time-option socket flag is time . The auxiliary function time_of_tltimeopt is used to do the conversion.

15.27 shutdown() (TCP and UDP)

$\text{shutdown} : (\text{fd} * \text{bool} * \text{bool}) \rightarrow \text{unit}$

A call of `shutdown(fd, r, w)` shuts down either the read-half of a connection, the write-half of a connection, or both. The `fd` is a file descriptor referring to the socket to shutdown; the `r` and `w` indicate whether the socket should be shut down for reading and writing respectively.

For a TCP socket, shutting down the read-half empties the socket's receive queue, but data will still be delivered to it and subsequent `recv()` calls will return data. Shutting down the write-half of a TCP connection causes the remaining data in the socket's send queue to be sent and then TCP's connection termination to occur.

For Linux and WinXP, a TCP socket may only be shut down if it is in the ESTABLISHED state; on FreeBSD a socket may be shut down in any state.

For a UDP socket, if the socket is shutdown for reading, data may still be read from the socket's receive queue on Linux, but on FreeBSD and WinXP this is not the case. Shutting down the socket for writing causes subsequent `send()` calls to fail.

15.27.1 Errors

A call to `shutdown()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|----------|--|
| ENOTCONN | The socket is not connected and so cannot be shut down. |
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| ENOBUFS | Out of resources. |

15.27.2 Common cases

A TCP socket is created and connects to a peer; data is transferred between the two; the socket has no more data to send so calls `shutdown()` to inform the peer of this: *socket_1; ...; connect_1; ...; shutdown_1; return_1*

15.27.3 API

Posix: `int shutdown(int socket, int how);`

FreeBSD: `int shutdown(int s, int how);`

Linux: `int shutdown(int s, int how);`

WinXP: `int shutdown(SOCKET s, int how);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to shut down. This corresponds to the `fd` argument of the model `shutdown()`.
- `how` is an integer specifying the type of shutdown corresponding to the (r, w) arguments in the model `shutdown()`. If `how` is set to `SHUT_RD` then the read half of the connection is to be shut down, corresponding to a `shutdown(fd, T, F)` call in the model; if it is set to `SHUT_WR` then the write half of the connection is to be shut down, corresponding to a `shutdown(fd, F, T)` call in the model; if it is set to `SHUT_RDWR` then both the read and write halves of the connection are to be shut down, corresponding to a `shutdown(fd, T, T)` call in the model.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux, and WinXP interfaces are similar, except where noted.

15.27.4 Model details

The following errors are not modelled:

- `EINVAL` signifies that the `how` argument is invalid. In the model the `how` argument is represented by the two boolean flags `r` and `w` which guarantees that the only values allowed are (T, T) , (T, F) , (F, T) , and

(**F**, **F**). The first three correspond to the allowed values of `how`: `SHUT_RD`, `SHUT_WR`, and `SHUT_RDWR`. The last possible value, (**F**, **F**), is not allowed by Posix, but the model allows a `shutdown(fd, F, F)` call, which has no effect on the socket.

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.27.5 Summary

| | | |
|-------------------------|--------------------------|---|
| <code>shutdown_1</code> | tcp: fast succeed | Shut down read or write half of TCP connection |
| <code>shutdown_2</code> | udp: fast succeed | Shutdown UDP socket for reading, writing, or both |
| <code>shutdown_3</code> | tcp: fast fail | Fail with <code>ENOTCONN</code> : cannot shutdown a socket that is not connected on Linux and WinXP |
| <code>shutdown_4</code> | udp: fast fail | Fail with <code>ENOTCONN</code> : socket's peer address not set on Linux |

15.27.6 Rules

| |
|--|
| <p><code>shutdown_1</code> tcp: fast succeed Shut down read or write half of TCP connection</p> $h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d); \text{socks} := \text{socks} \oplus [(sid, sock)] \rrbracket \xrightarrow{tid \cdot \text{shutdown}(fd, r, w)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}); \text{socks} := \text{socks} \oplus [(sid, sock')] \rrbracket$ <p> $sock = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \text{cantsndmore}, \text{cantrcvmore}, pr) \wedge$ $fd \in \text{dom}(h.fds) \wedge$ $fid = h.fds[fd] \wedge$ $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$ $pr = \text{TCP_PROTO } tcp_sock \wedge$ if <code>bsd_arch</code> $h.arch \wedge tcp_sock.st \in \{\text{CLOSED}; \text{LISTEN}\} \wedge w$ then </p> <p style="padding-left: 2em;"> let $sock'' = (\text{tcp_close } h.arch \text{ sock})$ in $sock' = sock'' \llbracket \text{cantsndmore} := (w \vee \text{cantsndmore});$ $\text{cantrcvmore} := (r \vee \text{cantrcvmore});$ $pr := \text{TCP_PROTO}(tcp_sock_of \text{ sock}'')$ $\llbracket cb := (\lambda cb.cb \llbracket \text{bsd_cantconnect} := \mathbf{T} \rrbracket);$ $lis := * \rrbracket$ </p> <p style="padding-left: 2em;"> else </p> <p style="padding-left: 4em;"> $(\neg \text{bsd_arch } h.arch \implies \exists i_1 i_2 p_2. tcp_sock.st = \text{ESTABLISHED} \wedge is_1 = \uparrow i_1 \wedge$ $ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2 \wedge tcp_sock.lis = *) \wedge$ $pr' = \text{TCP_PROTO}(tcp_sock \llbracket rcvq := [] \text{ onlywhen } r;$ $cb := (\lambda cb.cb \llbracket$ $\text{tf_shouldacknow} := \mathbf{T} \text{ onlywhen } w \rrbracket)) \rrbracket \wedge$ $sock' = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, w \vee \text{cantsndmore}, r \vee \text{cantrcvmore}, pr')$ </p> |
|--|

Description

From thread `tid`, which is in the `RUN` state, a `shutdown(fd, r, w)` call is made. `fd` refers to a TCP socket `sid` which is in the `ESTABLISHED` state and has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$.

The call succeeds: a `tid.shutdown(fd, r, w)` transition is made, leaving the thread in state `RET(OK())`. If $r = \mathbf{T}$ then the read-half of the connection is shut down, setting `cantrcvmore` = **T** and emptying the socket's receive queue; if $w = \mathbf{T}$ then the write-half of the connection is shut down, setting `cantsndmore` = **T**; otherwise, the socket is unchanged.

Variations

| | |
|---------|--|
| FreeBSD | The TCP socket can be in any state, not just ESTABLISHED. If the socket is in the CLOSED or LISTEN and is to be shutdown for writing, $w = \mathbf{T}$, then the socket is closed, see tcp_close (p121). Note that testing has shown the socket's listen queue is not always set to * after a shutdown() call. The precise condition for this being done needs to be investigated. |
|---------|--|

shutdown_2 **udp: fast succeed** Shutdown UDP socket for reading, writing, or both

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle; \\
 socks := socks \oplus \\
 \quad [(sid, sock \langle cantrcvmore := cantrcvmore; \\
 \quad cantsndmore := cantsndmore; \\
 \quad pr := \text{UDP_PROTO}(udp_pr) \rangle \rangle] \\
 \hline
 tid \cdot \text{shutdown}(fd, r, w) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched_timer}}) \rangle; \\
 socks := socks \oplus \\
 \quad [(sid, sock \langle cantrcvmore := (r \vee cantrcvmore); \\
 \quad cantsndmore := (w \vee cantsndmore); \\
 \quad pr := \text{UDP_PROTO}(udp_pr) \rangle \rangle]
 \end{array}$$

$$\begin{array}{l}
 fd \in \mathbf{dom}(h.fds) \wedge \\
 fid = h.fds[fd] \wedge \\
 h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 (\text{linux_arch } h.arch \implies sock.is_2 \neq *)
 \end{array}$$

Description

Consider a UDP socket sid , referenced by fd . From thread tid , which is in the RUN state, a $\text{shutdown}(fd, r, w)$ call is made and succeeds.

A $tid \cdot \text{shutdown}(fd, r, w)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. If the socket was shutdown for reading when the call was made or $r = \mathbf{T}$ then the socket is shutdown for reading. If the socket was shutdown for writing when the call was made or $w = \mathbf{T}$ then the socket is shutdown for writing.

Variations

| | |
|-------|--|
| Linux | As above, with the added condition that the socket's peer IP address must be set: $sock.is_2 \neq *$. |
|-------|--|

shutdown_3 **tcp: fast fail** Fail with ENOTCONN: cannot shutdown a socket that is not connected on Linux and WinXP

$$\begin{array}{l}
 h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \\
 \hline
 tid \cdot \text{shutdown}(fd, r, w) \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOTCONN}))_{\text{sched_timer}}) \rangle
 \end{array}$$

$$\begin{array}{l}
 fd \in \mathbf{dom}(h.fds) \wedge \\
 fid = h.fds[fd] \wedge \\
 h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
 \text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge \\
 tcp_sock.st \neq \text{ESTABLISHED} \wedge
 \end{array}$$

$\neg(\text{bsd_arch } h.\text{arch})$

Description

From thread tid , which is in the RUN state, a $\text{shutdown}(fd, r, w)$ call is made where fd refers to a TCP socket sid which is not in the ESTABLISHED state. The call fails with an ENOTCONN error.

A $tid.\text{shutdown}(fd, r, w)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOTCONN})$.

Variations

| | |
|---------|---------------------------|
| FreeBSD | This rule does not apply. |
|---------|---------------------------|

shutdown_4 **udp: fast fail Fail with ENOTCONN: socket's peer address not set on Linux**

$h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle;$

$socks := socks \oplus$

$[(sid, sock \langle is_2 := *; pr := \text{UDP_PROTO}(udp) \rangle)]$

$\underline{tid.\text{shutdown}(fd, r, w)}$

$\rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOTCONN}))_{\text{sched_timer}}) \rangle;$

$socks := socks \oplus$

$[(sid, sock \langle is_2 := *;$

$cantsndmore := (w \vee sock.cantsndmore);$

$cantrcvmore := (r \vee sock.cantrcvmore);$

$pr := \text{UDP_PROTO}(udp) \rangle)]$

$\text{linux_arch } h.\text{arch} \wedge$

$fd \in \text{dom}(h.\text{fds}) \wedge$

$fid = h.\text{fds}[fd] \wedge$

$h.\text{files}[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff)$

Description

On Linux, consider a UDP socket sid referenced by fd with no peer IP address set: $is_2 := *$. From thread tid , which is in the RUN state, a $\text{shutdown}(fd, r, w)$ call is made, and fails with an ENOTCONN error.

A $tid.\text{shutdown}(fd, r, w)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOTCONN})$. If the socket was shutdown for reading when the call was made or $r = \mathbf{T}$ then the socket is shutdown for reading. If the socket was shutdown for writing when the call was made or $w = \mathbf{T}$ then the socket is shutdown for writing.

Variations

| | |
|---------|--|
| FreeBSD | This rule does not apply: see rule <i>shutdown_2</i> . |
| WinXP | This rule does not apply: see rule <i>shutdown_2</i> . |

15.28 socketmark() (TCP only)

$\text{socketmark} : fd \rightarrow \text{bool}$

A call to $\text{socketmark}(fd)$ returns a bool specifying whether or not a socket is at the urgent mark. Here fd is a file descriptor referring to a socket.

If fd refers to a TCP socket then the call will succeed, returning \mathbf{T} if that socket is at the urgent mark, and \mathbf{F} if it is not.

If `fd` refers to a UDP socket then on FreeBSD the call will return **F** and on all other architectures it will fail with an `EINVAL` error: there is no concept of urgent data for UDP so calling `socketatmark()` does not make sense.

15.28.1 Errors

A call to `socketatmark()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|-----------------------|--|
| <code>EINVAL</code> | Calling <code>socketatmark()</code> on a UDP socket does not make sense. |
| <code>EBADF</code> | The file descriptor passed is not a valid file descriptor. |
| <code>ENOTSOCK</code> | The file descriptor passed does not refer to a socket. |

15.28.2 Common cases

socketatmark_1; return_1

15.28.3 API

```
Posix:      int socketatmark(int s);
FreeBSD:   int ioctl(int d, unsigned long request, int* argp);
Linux:     int ioctl(int d, int request, int* argp);
WinXP:     int ioctlsocket(SOCKET s, long cmd, u_long* argp);
In the Posix interface:
```

- `s` is a file descriptor referring to a socket. This corresponds to the `fd` argument of the model `socketatmark()`.
- the returned `int` is either 0 or 1 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. If the return value is 1 then the socket is at the urgent mark corresponding to a return value of **T** in the model `socketatmark()`; if the return value is 0 then the socket is not at the urgent mark, corresponding to a return value of **F** in the model.

The FreeBSD, Linux, and WinXP interfaces are significantly different: to check whether or not a socket is at the urgent mark, the `ioctl()` function must be used. In the FreeBSD interface:

- `d` is a file descriptor referring to a socket, corresponding to the `fd` argument of the model `socketatmark()`.
- `request` selects which control function is to be performed. For `socketatmark()`, the request is `SIOCATMARK`.
- `argp` is a pointer to a location to store the result of the call in. If the socket is at the urgent mark then 1 will be in the location pointed to by `argp` upon return, corresponding to a return value of **T** in the model `socketatmark()`; if the socket is not at the urgent mark, then `argp` will contain the value 0, corresponding to a return value of **F** in the model.
- the returned `int` is either 0 to indicate success or -1 to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not -1, with the actual error code available through a call to `WSAGetLastError()`.

The Linux and WinXP interfaces are similar.

15.28.4 Model details

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, `EFAULT` can be returned if the `argp` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model `socketatmark()`.
- On FreeBSD and Linux, `EINVAL` can be returned if `request` is not a valid request. The model `socketatmark()` is implemented using the `SIOCATMARK` request which is valid.

- ENOTTY is possible when making an *ioctl()* call but is not modelled.
- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.28.5 Summary

| | | |
|---------------------|--------------------------|--|
| <i>socketmark_1</i> | tcp: fast succeed | Successfully return whether or not a TCP socket is at the urgent mark |
| <i>socketmark_2</i> | udp: rc | Fail with EINVAL: calling socketmark() on a UDP socket does not make sense |

15.28.6 Rules

socketmark_1 **tcp: fast succeed** Successfully return whether or not a TCP socket is at the urgent mark

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid \cdot \text{socketmark}(fd)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } b))_{\text{sched_timer}}) \rrbracket$$

$$\begin{aligned} &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \text{cantsndmore}, \text{cantrcvmore}, \\ &\quad \text{TCP_Sock}(\text{ESTABLISHED}, cb, *, \text{sndq}, \text{sndurp}, \text{rcvq}, \text{rcvurp}, iobc)) \wedge \\ &b = (\text{rcvurp} = \uparrow 0) \end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a *socketmark(fd)* call is made. *fd* refers to a TCP socket identified by *sid* which is in the ESTABLISHED state and has binding quad ($\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2$). The call succeeds, returning **T** if the socket is at the urgent mark: *rcvurp* = $\uparrow 0$; or **F** otherwise.

A *tid*·*socketmark(fd)* transition is made, leaving the thread state *RET(OK b)* where *b* is a boolean: **T** or **F** as above.

socketmark_2 **udp: rc** Fail with EINVAL: calling socketmark() on a UDP socket does not make sense

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid \cdot \text{socketmark}(fd)} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched_timer}}) \rrbracket$$

$$\begin{aligned} &\text{proto_of}(h.socks[sid]).pr = \text{PROTO_UDP} \wedge \\ &fd \in \mathbf{dom}(h.fds) \wedge \\ &fid = h.fds[fd] \wedge \\ &h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\ &\mathbf{if} \text{ bsd_arch } h.arch \mathbf{then} rc = \text{FAST SUCCEED} \wedge ret = \text{OK}(\mathbf{F}) \\ &\mathbf{else} rc = \text{FAST FAIL} \wedge ret = \text{FAIL EINVAL} \end{aligned}$$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the RUN state, a *socketmark(fd)* call is made. On FreeBSD the call succeeds, returning **F**; on Linux and WinXP the call fails with an EINVAL error.

A *tid*·*socketmark(fd)* transition is made, leaving the thread state *RET(OK(F))* on FreeBSD, and in state *RET(FAIL EINVAL)* on Linux and WinXP.

Variations

| | |
|---------|---|
| Posix | As above: the call succeeds, returning F . |
| FreeBSD | As above: the call succeeds, returning F . |
| Linux | As above: the call fails with an EINVAL error. |
| WinXP | As above: the call fails with an EINVAL error. |

15.29 socket() (TCP and UDP)

`socket` : *sock_type* → fd

A call to `socket(type)` creates a new socket. Here *type* is the type of socket to create: `SOCK_STREAM` for TCP and `SOCK_DGRAM` for UDP. The returned fd is the file descriptor of the new socket.

15.29.1 Errors

A call to `socket()` can fail with the errors below, in which case the corresponding exception is raised:

| | |
|---------|--|
| EMFILE | No more file descriptors for this process. |
| ENOBUFS | Out of resources. |
| ENOMEM | Out of resources. |
| ENFILE | Out of resources. |

15.29.2 Common cases

TCP: *socket_1*; *return_1*; *connect_1*; ... UDP: *socket_1*; *return_1*; *bind_1*; *return_1*; *send_9*; ...

15.29.3 API

```
Posix:    int socket(int domain, int type, int protocol);
FreeBSD:  int socket(int domain, int type, int protocol);
Linux:    int socket(int doamin, int type, int protocol);
WinXP:    SOCKET socket(int af, int type, int protocol);
```

In the Posix interface:

- **domain** specifies the communication domain in which the socket is to be created, specifying the protocol family to be used. Only IPv4 sockets are modelled here, so **domain** is set to `AF_INET` or `PF_INET`.
- **type** specifies the communication semantics: `SOCK_STREAM` provides sequenced, reliable, two-way, connection-based byte streams; `SOCK_DGRAM` supports datagrams (connectionless, unreliable messages of a fixed maximum length). This corresponds to the *sock_type* argument of the model `socket()`.
- **protocol** specifies the particular protocol to be used for the socket. A **protocol** of 0 requests to use the default for the appropriate socket **type**: TCP for `SOCK_STREAM` and UDP for `SOCK_DGRAM`. Alternatively a specific protocol number can be used: 6 for TCP and 17 for UDP. In the model, `SOCK_STREAM` refers to a TCP socket and `SOCK_DGRAM` to a UDP socket so the **protocol** argument is not necessary.

A call to `socket(SOCK_STREAM)` in the model interface, would be a `socket(AF_INET, SOCK_STREAM, 0)` call in Posix; a call to `socket(SOCK_DGRAM)` in the model interface would be a `socket(AF_INET, SOCK_DGRAM, 0)` call in Posix.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.29.4 Model details

The following errors are not modelled:

- In Posix and on Linux, `EACCES` specifies that the process does not have appropriate privileges. We do not model a privilege state in which socket creation would be disallowed.
- In Posix and on Linux, `EAFNOSUPPORT`, specifies that the implementation does not support the address domain. FreeBSD, Linux, and WinXP all support `AF_INET` sockets.
- On Linux, `EINVAL` means unknown protocol, or protocol domain not available. Both TCP and UDP are known protocols for Linux, and `AF_INET` is a known domain on Linux.
- In Posix and on Linux, `EPROTONOTSUPPORT` specifies that the protocol is not supported by the address family, or the protocol is not supported by the implementation. FreeBSD, Linux, and WinXP all support the TCP and UDP protocols.
- In Posix, `EPROTOTYPE` signifies that the socket type is not supported by the protocol. Both `SOCK_STREAM` and `SOCK_DGRAM` are supported by TCP and UDP respectively.
- On WinXP, `WSAESOCKTNOSUPPORT` means the specified socket type is not supported in this address family. The `AF_INET` family supports both `SOCK_STREAM` and `SOCK_DGRAM` sockets.

The `AF_INET6`, `AF_LOCAL`, `AF_ROUTE`, and `AF_KEY` address families; `SOCK_RAW` socket type; and all protocols other than TCP and UDP are not modelled.

15.29.5 Summary

| | | |
|-----------------------|--------------------------|--|
| <code>socket_1</code> | all: fast succeed | Successfully return a new file descriptor for a fresh socket |
| <code>socket_2</code> | all: fast fail | Fail with <code>EMFILE</code> : out of file descriptors for this process |

15.29.6 Rules

`socket_1` **all: fast succeed** Successfully return a new file descriptor for a fresh socket

$$\begin{array}{l} h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ fds := fds; \\ files := files; \\ socks := socks \rangle \\ \underline{tid \cdot (\text{socket}(\text{sockettype}))} \rightarrow h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK } fd))_{\text{sched_timer}}); \\ fds := fds'; \\ files := files \oplus [(fid, \text{FILE}(\text{FT_SOCKET}(sid), \text{ff_default}))]; \\ socks := socks \oplus [(sid, sock)] \rangle \end{array}$$

card(**dom**(*fds*)) < OPEN_MAX \wedge
fid \notin (**dom**(*files*)) \wedge
sid \notin (**dom**(*socks*)) \wedge
nextfd *h.arch fds fd* \wedge
fds' = *fds* \oplus (*fd*, *fid*) \wedge
(case sockettype of
SOCK_DGRAM \rightarrow (*sock* =
SOCK(\uparrow *fid*, sf_default *h.arch sockettype*, *, *, *, *, *, **F**, **F**, UDP_Sock([])) ||
SOCK_STREAM \rightarrow (*sock* =
SOCK(\uparrow *fid*, sf_default *h.arch sockettype*, *, *, *, *, *, **F**, **F**,

TCP_Sock(CLOSED, initial_cb, *, [], *, [], *, NO_OOBDATA))))

Description

From thread tid , which is in the RUN state, a `socket(socktype)` call is made. The number of open file descriptors is less than the maximum permitted, OPEN_MAX.

If $socktype = SOCK_STREAM$ then a new TCP socket $sock$ is created, in the CLOSED state, with `initial_cb` (p101) as its control block, and all other fields uninitialised; if $socktype = SOCK_DGRAM$ then a new, uninitialised UDP socket $sock$ is created. A new open file description is created pointing to the socket, and a new file descriptor, fd , is allocated in an architecture specific way (see `nextfd` (p??)) to point to the open file description. The host's finite map of sockets is updated to include an entry mapping the socket identifier sid to the socket; its finite map of file descriptions is updated to add an entry mapping the file descriptor fd to the file description of the socket; and its finite map of file descriptors is updated, adding a mapping from fd to fd .

A $tid \cdot \text{socket}(sock_type)$ transition is made, leaving the thread state `RET(OKfd)` to return the new file descriptor.

socket_2 **all: fast fail** Fail with EMFILE: out of file descriptors for this process

$$h \langle \langle ts := ts \oplus (tid \mapsto (RUN)_d) \rangle \rangle \xrightarrow{tid \cdot (\text{socket}(s))} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EMFILE}))_{\text{sched_timer}}) \rangle \rangle$$

$\text{card}(\text{dom}(h.fds)) \geq \text{OPEN_MAX}$

Description

From thread tid , which is in the RUN state, a `socket(s)` call is made. The number of open file descriptors is greater than the maximum allowed number, OPEN_MAX, and so the call fails with an EMFILE error.

A $tid \cdot \text{socket}(s)$ transition is made, leaving the thread state `RET(FAIL EMFILE)`.

15.30 Miscellaneous (TCP and UDP)

This section collects the remaining Sockets API rules:

- The rule *return_1* characterising how the the results of system calls are returned to the caller, with transitions from the thread state `(RET v)d`.
- Rules *badf_1* and *notsock_1* deal with all the Sockets API calls that take a file descriptor argument, dealing uniformly with the error cases in which that file descriptor is not valid or does not refer to a socket.
- Rule *intr_1* applies to all the thread states for blocked calls, `ACCEPT2(sid)` etc., characterising the behaviour in the case where the call is interrupted by a signal.
- Rules *resourcefail_1* and *resourcefail_2* deal with the cases where calls fail due to a lack of system resources.

15.30.1 Errors

Common errors.

| | |
|----------|--|
| EBADF | The file descriptor passed is not a valid file descriptor. |
| ENOTSOCK | The file descriptor passed does not refer to a socket. |
| EINTR | The system was interrupted by a caught signal. |

| | |
|---------|-------------------|
| ENOMEM | Out of resources. |
| ENOBUFS | Out of resources. |
| ENFILE | Out of resources. |

15.30.2 Summary

| | | |
|-----------------------|-------------------------------------|---|
| <i>return_1</i> | all: misc nonurgent | Return result of system call to caller |
| <i>badf_1</i> | all: fast fail | Fail with EBADF: not a valid file descriptor |
| <i>notsock_1</i> | all: fast fail | Fail with ENOTSOCK: file descriptor not a valid socket |
| <i>intr_1</i> | all: slow nonurgent fail | Fail with EINTR: blocked system call interrupted by signal |
| <i>resourcefail_1</i> | all: fast badfail | Fail with ENFILE, ENOBUFS or ENOMEM: out of resources |
| <i>resourcefail_2</i> | all: slow nonurgent bad-fail | Fail with ENFILE, ENOBUFS or ENOMEM: from a blocked state with out of resources |

15.30.3 Rules

return_1 **all: misc nonurgent** Return result of system call to caller

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET } v)_d) \rangle \rangle \xrightarrow{tid \cdot v} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_{\text{never_timer}}) \rangle \rangle$$

T

Description

A system call from thread *tid* has completed, leaving the thread state $(\text{RET } v)_d$. The value *v* (which may be of the form OK *v'* or FAIL *v'*, for success or failure respectively) is returned to the caller before the timer *d* expires. The thread continues its execution, indicated by the resulting thread state $(\text{RUN})_{\text{never_timer}}$.

badf_1 **all: fast fail** Fail with EBADF: not a valid file descriptor

$$h \langle \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \rangle \xrightarrow{tid \cdot \text{opn}} h \langle \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rangle \rangle$$

$fd_op \text{ } fd \text{ } \text{opn} \wedge$

$fd \notin \text{dom}(h.fds) \wedge$

(if windows_arch *h.arch* then $e = \text{ENOTSOCK}$ else $e = \text{EBADF}$)

Description

From thread *tid*, which is in the RUN state, a system call *opn* is made. The call requires a single valid file descriptor, but the descriptor passed, *fd* is not valid: it does not refer to an open file description. The call fails with an EBADF error, or an ENOTSOCK error on WinXP.

A *tid.opn* transition is made, leaving the thread state $\text{RET}(\text{FAIL } e)$ where *e* is one of the above errors.

The system calls this rule applies to are: `accept()`, `bind()`, `close()`, `connect()`, `disconnect()`, `dup()`, `dupfd()`, `getfileflags()`, `setfileflags()`, `getsockname()`, `getpeername()`, `getsockbopt()`, `getsockerr()`, `getsocklistening()`, `getsocknopt()`, `getsocktopt()`, `listen()`, `recv()`, `send()`, `setsockbopt()`, `setsocknopt()`, `setsocktopt()`, `shutdown()`, and `socketatmark()`. See the definition of `fd_op` (p35).

Variations

| | |
|---------|--|
| FreeBSD | As above: the call fails with an EBADF error. |
| Linux | As above: the call fails with an EBADF error. |
| WinXP | As above: the call fails with an ENOTSOCK error. |

notsock_1 **all: fast fail** Fail with ENOTSOCK: file descriptor not a valid socket

$$h \langle\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle\rangle \xrightarrow{tid \cdot \text{opn}} h \langle\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOTSOCK}))_{\text{sched_timer}}) \rangle\rangle$$

$fd_sockop \text{ } fd \text{ } opn \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fid] = \text{FILE}(ft, ff) \wedge$
 $\neg(\exists sid.ft = \text{FT_SOCKET}(sid))$

Description

From thread *tid*, which is in the RUN state, a system call *opn* is made. The call requires a single file descriptor referring to a socket. The file descriptor *fd* that the user passes refers to an open file description $\text{FILE}(ft, ff)$ that does not refer to a socket. The call fails with an ENOTSOCK error.

A *tid.opn* transition is made, leaving the thread state $\text{RET}(\text{FAIL ENOTSOCK})$.

The system calls this rule applies to are: `accept()`, `bind()`, `connect()`, `disconnect()`, `getpeername()`, `getsockbopt()`, `getsockerr()`, `getsocklistening()`, `getsockname()`, `getsocknopt()`, `getsocktopt()`, `listen()`, `recv()`, `send()`, `setsockbopt()`, `setsocknopt()`, `setsocktopt()`, `shutdown()`, and `socketmark()`. See the definition of `fd_sockop` (p35).

intr_1 **all: slow nonurgent fail** Fail with EINTR: blocked system call interrupted by signal

$$h \langle\langle ts := ts \oplus (tid \mapsto (st)_d) \rangle\rangle \xrightarrow{\tau} h \langle\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINTR}))_{\text{sched_timer}}) \rangle\rangle$$

$sock = (h.socks[sid]) \wedge$
 $(st = \text{CLOSE2}(sid) \vee$
 $st = \text{CONNECT2}(sid) \vee$
 $st = \text{RCV2}(sid, n, opts) \vee$
 $st = \text{SEND2}(sid, addr, str, opts) \vee$
 $st = \text{PSELECT2}(readfds, writefds, exceptfds) \vee$
 $st = \text{ACCEPT2}(sid))$

Description

If on socket *sid* as user call blocked leaving a thread in one of the states: $\text{CLOSE2}(sid)$, $\text{CONNECT2}(sid)$, $\text{RCV2}(sid)$, $\text{SEND2}(sid)$, $\text{PSELECT2}(sid)$ or $\text{ACCEPT2}(sid)$ and a signal is caught, the calls fails returning error EINTR.

Model details

This rule is non-deterministic, allowing blocked calls to be interrupted at any point, as the specification does not model the dynamics of signals.

Variations

| | |
|-------|--|
| POSIX | POSIX says that a system call "shall fail" if "interrupted by a signal". |
|-------|--|

resourcefail_1 **all: fast badfail** Fail with ENFILE, ENOBUFS or ENOMEM: out of resources

$$h \llbracket ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rrbracket \xrightarrow{tid \cdot call} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rrbracket$$

$\neg \text{INFINITE_RESOURCES} \wedge$
 $fd \in \mathbf{dom}(h.fds) \wedge$
 $fid = h.fds[fd] \wedge$
 $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
 $sock = (h.socks[sid]) \wedge$
 $((call = \text{socket}(socktype) \wedge e \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\}) \vee$
 $(call = \text{bind}(fd, is_1, ps_1) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{connect}(fd, i_2, \uparrow p_2) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{listen}(fd, n) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{recv}(fd, n, opts) \wedge e \in \{\text{ENOMEM}; \text{ENOBUFS}\}) \vee$
 $(call = \text{getsockname}(fd) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{getpeername}(fd) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{shutdown}(fd, r, w) \wedge e = \text{ENOBUFS}) \vee$
 $(call = \text{accept}(fd) \wedge e \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\})$
 $\wedge \text{proto_of } sock.pr = \text{PROTO_TCP})$

Description

Thread *tid* performs a `socket()`, `bind()`, `connect()`, `listen()`, `recv()`, `getsockname()`, `getpeername()`, `shutdown()` or `accept()` system call on socket *sid*, referred to by *fd*, when insufficient system-wide resources are available to complete the request. Return a failure of ENFILE, ENOBUFS or ENOMEM immediately to the calling thread.

This rule applies only when it is assumed that the host being modelled does not have INFINITE_RESOURCES, i.e. the host does not have unlimited memory, mbufs, file descriptors, etc.

Model details

The modelling of failure is deliberately non-deterministic because the cause of errors such as ENFILE are determined by more than is modelled in this specification. In order to be more precise, the model would need to describe the whole system to determine when such error conditions could and should arise.

resourcefail_2 **all: slow nonurgent badfail** Fail with ENFILE, ENOBUFS or ENOMEM: from a blocked state with out of resources

$$h \llbracket ts := ts \oplus (tid \mapsto (t)_d) \rrbracket \xrightarrow{\tau} h \llbracket ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched_timer}}) \rrbracket$$

$\neg \text{INFINITE_RESOURCES} \wedge$
 $sock = (h.socks[sid]) \wedge$
 $((t = \text{ACCEPT2}(sid) \wedge e \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\}) \vee$
 $(t = \text{CONNECT2}(sid) \wedge e = \text{ENOBUFS}) \vee$
 $(t = \text{RCV2}(sid, n, opts) \wedge e \in \{\text{ENOBUFS}; \text{ENOMEM}\}))$

Description

If thread *tid* of host *h* is in state `ACCEPT2(sid)`, `CONNECT2(sid)` or `RCV2(sid)` following an `accept()`, `connect()` or `recv()` system call that blocked, and the host has subsequently exhausted its system-wide resources, fail with ENFILE, ENOBUFS or ENOMEM. The error is immediately returned to the thread that made the system call.

Calls to `connect()` only return ENOBUFS when resources are exhausted and calls to `recv()` only return ENOBUFS or ENOMEM.

This rule applies only when it is assumed that the host being modelled does not have INFINITE_RESOURCES, i.e. the host does not have unlimited memory, mbufs, file descriptors, etc.

Model details

The modelling of failure is deliberately non-deterministic because the cause of errors such as ENFILE are determined by more than is modelled in this specification. In order to be more precise, the model would need to describe the whole system to determine when such error conditions could and should arise.

Chapter 16

Host LTS: TCP Input Processing

16.1 Input Processing (TCP only)

These rules deal with the processing of TCP segments from the host's input queue. The most important are *deliver_in_1*, *deliver_in_2*, and *deliver_in_3*.

deliver_in_1 deals with a passive open: a socket in LISTEN state that receives a *SYN* and sends a *SYN, ACK*.

deliver_in_2 deals with the completion of an active open: a socket in SYN_SENT state (that has previously sent a *SYN* with the *connect_1* rule) that receives a *SYN, ACK* and sends an *ACK*. It also deals with simultaneous opens.

deliver_in_3 deals with the common cases of TCP data exchange and connection close: sockets in connected states that receive data, *ACKs*, and *FINs*. This rule is structured using the relational monad, combining auxiliaries *di3_topstuff*, *di3_ackstuff*, *di3_datastuff* etc., to factor out many of the imperative effects of the code.

The other rules deal with *RSTs* and a variety of pathological situations.

16.1.1 Summary

| | | |
|-----------------------------|-------------------------------|--|
| <i>deliver_in_1</i> | tcp: network nonurgent | Passive open: receive SYN, send SYN,ACK |
| <i>deliver_in_1b</i> | tcp: network nonurgent | For a listening socket, receive and drop a bad datagram and either generate a RST segment or ignore it. Drop the incoming segment if the socket's queue of incomplete connections is full. |
| <i>deliver_in_2</i> | tcp: network nonurgent | Completion of active open (in SYN_SENT receive SYN,ACK and send ACK) or simultaneous open (in SYN_SENT receive SYN and send SYN,ACK) |
| <i>deliver_in_2a</i> | tcp: network nonurgent | Receive bad or boring datagram and RST or ignore for SYN_SENT socket |
| <i>deliver_in_3</i> | tcp: network nonurgent | Receive data, FINs, and ACKs in a connected state |
| <i>di3_topstuff</i> | | <i>deliver_in_3</i> initial checks |
| <i>di3_newackstuff</i> | | <i>deliver_in_3</i> new ack processing, used in <i>di3_ackstuff</i> |
| <i>di3_ackstuff</i> | | <i>deliver_in_3</i> ACK processing |
| <i>di3_datastuff_really</i> | | <i>deliver_in_3</i> data processing |
| <i>di3_datastuff</i> | | <i>deliver_in_3</i> data processing |
| <i>di3_ststuff</i> | | <i>deliver_in_3</i> TCP state change processing |
| <i>di3_socks_update</i> | | <i>deliver_in_3</i> socket update processing |
| <i>deliver_in_3a</i> | tcp: network nonurgent | Receive data with invalid checksum or offset |
| <i>deliver_in_3b</i> | tcp: network nonurgent | Receive data after process has gone away |
| <i>deliver_in_3c</i> | tcp: network nonurgent | Receive stupid ACK or LAND DoS in SYN_RECEIVED state |
| <i>deliver_in_4</i> | tcp: network nonurgent | Receive and drop (silently) a non-sane or martian segment |
| <i>deliver_in_5</i> | tcp: network nonurgent | Receive and drop (maybe with RST) a sane segment that does not match any socket |

| | | |
|----------------------|-------------------------------|---|
| <i>deliver_in_6</i> | tcp: network nonurgent | Receive and drop (silently) a sane segment that matches a CLOSED socket |
| <i>deliver_in_7</i> | tcp: network nonurgent | Receive RST and zap non-{CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} socket |
| <i>deliver_in_7a</i> | tcp: network nonurgent | Receive RST and zap SYN_RECEIVED socket |
| <i>deliver_in_7b</i> | tcp: network nonurgent | Receive RST and ignore for LISTEN socket |
| <i>deliver_in_7c</i> | tcp: network nonurgent | Receive RST and ignore for SYN_SENT(unacceptable ack) or TIME_WAIT socket |
| <i>deliver_in_7d</i> | tcp: network nonurgent | Receive RST and zap SYN_SENT(acceptable ack) socket |
| <i>deliver_in_8</i> | tcp: network nonurgent | Receive SYN in non-{CLOSED; LISTEN; SYN_SENT; TIME_WAIT} state |
| <i>deliver_in_9</i> | tcp: network nonurgent | Receive SYN in TIME_WAIT state if there is no matching LISTEN socket or sequence number has not increased |

16.1.2 Rules

deliver_in_1 **tcp: network nonurgent** **Passive open: receive SYN, send SYN,ACK**

$h \langle \{socks := socks \oplus [(sid, sock)];$
 $iq := iq;$
 $oq := oq \rangle$

$\xrightarrow{\tau}$

$h \langle \{socks := socks' \oplus$
 (* Listening socket *)
 $[(sid, SOCK(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
 $TCP_Sock(LISTEN, cb, \uparrow lis', [], *, [], *, NO_OOBDATA))];$
 (* New socket formed by the incoming SYN *)
 $(sid', SOCK(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore,$
 $TCP_Sock(SYN_RECEIVED, cb'', *, [], *, [], *, NO_OOBDATA))];$
 $iq := iq';$
 $oq := oq' \rangle$

(* **Summary:** A host h with listening socket $sock$ referenced by index sid receives a valid and well-formed SYN segment seg addressed to socket $sock$. A new socket in the SYN_RECEIVED state is constructed, referenced by sid' ($\neq sid$), is added to the queue of incomplete incoming connection attempts q , and a SYN,ACK segment is generated in reply with some field values being chosen or negotiated. The reply segment is finally queued on the host's output queue for transmission, ignoring any errors upon queueing failure. *)

$sid \notin (\mathbf{dom}(socks)) \wedge$
 $sid' \notin (\mathbf{dom}(socks)) \wedge$
 $sid \neq sid' \wedge$

(* Take TCP segment seg from the head of the host's input queue *)
 $\text{dequeue_iq}(iq, iq', \uparrow(TCP\ seg)) \wedge$

(* The segment must be of an acceptable form *)

(* Note: some segment fields are ignored during TCP connection establishment and as such may contain arbitrary values. These are equal to the identifiers postfixed with $_discard$ below, which are otherwise unconstrained. *)
 $(\exists win_ws_mss_PSH_discard\ URG_discard\ FIN_discard\ urp_discard\ data_discard\ ack_discard.$

$seg =$
 $\langle \{ is_1 := \uparrow i_2;$
 $is_2 := \uparrow i_1;$
 $ps_1 := \uparrow p_2;$
 $ps_2 := \uparrow p_1;$
 $seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);$
 $ack := tcp_seq_flip_sense(ack_discard : tcp_seq_local);$


```

    URG := URG_discard;
    ACK := F; (* ACK must be F in a SYN segment *)
    PSH := PSH_discard;
    RST := F; (* Valid SYN segments never have RST set *)
    SYN := T; (* Is a SYN segment! *)
    FIN := FIN_discard;
    win := win_;
    ws := ws_;
    urp := urp_discard;
    mss := mss_;
    ts := ts;
    data := data_discard
  } ∧

```

```

(* Equality of some type casts *)
w2n win_ = win ∧
option_map ord ws_ = ws ∧
option_map w2n mss_ = mss
) ∧

```

```

(* The segment is addressed to an IP address belonging to one of the interfaces of host h and is not addressed from or
to a link-layer multicast or an IP-layer broadcast address *)
i1 ∈ local_ips h.ifds ∧
¬(is_broadormulticast h.ifds i1) ∧
¬(is_broadormulticast h.ifds i2) ∧

```

```

(* Find the socket sock that has the best match for the address quad in segment seg, see tcp_socket_best_match (p86).
Socket sock must have a form matching the patten SOCK(...). *)
tcp_socket_best_match socks(sid, sock) seg h.arch ∧
sock = SOCK(↑ fid, sf, is1, ↑ p1, is2, ps2, es, cantsndmore, cantrcvmore,
TCP_Sock(LISTEN, cb, ↑ lis, [], *, [], *, NO_OOBDATA)) ∧

```

```

(* A BSD socket in the LISTEN state may have its peer's IP address is2 and port ps2 set because listen() can be
called from any TCP state. On other architectures they are both constrained to *. *)
((is2 = * ∧ ps2 = *) ∨
(bsd_arch h.arch ∧ is2 = ↑ i2 ∧ ps2 = ↑ p2)) ∧

```

```

(* If socket sid has a local IP address specified it should be the same as the destination IP address of the segment
seg, otherwise the seg is not addressed to this socket. If the socket does not have a local IP address the segment is
acceptable because the socket is listening on all local IP addresses. The segment must not have been sent by socket
sock. Note: a socket is permitted to connect to itself by a simultaneous open. This is handled by deliver_in_2 (p285)
and not here. *)
(case is1 of ↑ il' → il' = i1 || * → T) ∧
¬(i1 = i2 ∧ p1 = p2) ∧

```

```

(* If another socket in the TIME_WAIT state matches the address quad of the SYN segment then only proceed with
the new incoming connection attempt if the sequence number of the segment seq is strictly greater than the next
expected sequence number on the TIME_WAIT socket, rcv_nxt. This prevents old or duplicate SYN segments from
previous incarnations of the connection from inadvertently creating new connections. *)
¬(∃(sid, sock) :: socks.
  ∃tcp_sock.
    sock.pr = TCP_PROTO(tcp_sock) ∧
    tcp_sock.st = TIME_WAIT ∧
    sock.is1 = ↑ i1 ∧ sock.ps1 = ↑ p1 ∧ sock.is2 = ↑ i2 ∧ sock.ps2 = ↑ p2 ∧
    seq ≤ tcp_sock.cb.rcv_nxt) ∧

```

```

(* Otherwise, the TIME_WAIT sock is completely defunct because there is a new connection attempt from the same
remote end-point. Close it completely. *)

```

(* Note: this models the behaviour in RFC1122 Section 4.2.2.13 which states that a new SYN with a sequence number larger than the maximum seen in the last incarnation may reopen the connection, i.e., reuse the socket for the new connection changing out of the TIME_WAIT state. This is modelled by closing the existing TIME_WAIT socket and creating the new socket from scratch. *)

```
socks' = $o_f(\lambda sock.
  if \exists tcp_sock.sock.pr = TCP_PROTO(tcp_sock) \wedge
    tcp_sock.st = TIME_WAIT \wedge
    sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge
    sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2
  then
    tcp_close h.arch sock
  else
    sock
)socks \wedge
```

(* Accept the new connection attempt to the incomplete connection queue if the queue of completed (established) connections is not already full *)
accept_incoming_q0 lis **T** \wedge

(* Possibly drop an arbitrary connection from the queue of incomplete connection attempts – this covers the behaviour of FreeBSD when the oldest connection in the SYN bucket or in the whole SYN cache is dropped, depending upon which became full. *)

```
(choose drop :: drop_from_q0 lis.
  if drop then
    \exists q0L sid'' q0R.
    lis.q0 = q0L @ (sid'' :: q0R) \wedge
    q'_0 = q0L @ q0R
  else
    q'_0 = lis.q0
) \wedge
```

(* Put the new incomplete connection on the (possibly pruned) incomplete connections queue. *)

```
lis' = lis \lfloor q_0 := sid' :: q'_0 \rfloor \wedge
```

(* Create a SYN,ACK segment in reply: *)

(* The maximum segment size of the outgoing SYN,ACK reply segment must be in range, i.e., less than the maximum IP segment size minus the space consumed by IP and TCP headers. This is deliberately non-deterministic: an implementation would query the interface's MTU and subtract the header space required. *)

```
advms \in \{n | n \ge 1 \wedge n \le (65535 - 40)\} \wedge
```

(* Be non-deterministic in deciding whether to transmit a maximum segment size option. A host either supports the maximum segment size option or not – here the specification permits either sending the option or not, but if the option is sent it must contain the advertised mss chosen previously by the host. This captures all acceptable behaviour. *)

```
advms' \in \{*\; \uparrow advms\} \wedge
```

(* If a timestamp option was present in the received segment and a non-deterministic choice is made to do timestamping on this connection (i.e., the host supports timestamping), then timestamping is being used for this connection. Otherwise, timestamping is not used because one or both hosts do not support it. A real host would either do timestamping or not depending on its configuration. Here all acceptable behaviour must be permitted. *)

```
tf_rcvd_tstmp' = is_some ts \wedge
(choose want_tstmp :: \{F; T\}.
  tf_doing_tstmp' = (tf_rcvd_tstmp' \wedge want_tstmp)
) \wedge
```

(* Lookup the bandwidth delay product from the route metric cache and calculate the size of the receive and send buffers, the maximum segment size and the initial congestion window. *)

```
bw_delay_product_for_rt = * \wedge
(rcvbufsize', sndbufsize', t_maxseg', snd_cwnd') =
```

calculate_buf_sizes *advms* *mss* *bw_delay_product_for_rt*(*is_localnet* *h.ifds* *i2*)
 (*sf.n*(SO_RCVBUF))(*sf.n*(SO_SNDBUF))*tf_doing_tstamp'* *h.arch* \wedge

(* Store the new receive and send buffer sizes *)

sf' = *sf* \llbracket *n* := funupd_list *sf.n*[(SO_RCVBUF, *rcvbufsize'*); (SO_SNDBUF, *sndbufsize'*)] \rrbracket \wedge

(* Non-deterministically choose to do window scaling (i.e., choose whether this host supports window scaling or not). Do window scaling on the new connection if the received SYN segment contained a window scaling option and this host supports it. A real host would either be configured to do window scaling or not (provided it supported window scaling). Here all acceptable behaviour must be permitted. *)

req_ws \in {**F**; **T**} \wedge

tf_doing_ws' = (*req_ws* \wedge **is_some** *ws*) \wedge

(**if** *tf_doing_ws'* **then** (* Doing window scaling *)

(* Constrain the receive scale to be within the correct range and the send scale to be that received from the remote host *)

rcv_scale' \in {*n* | *n* \geq 0 \wedge *n* \leq TCP_MAXWINSIZE} \wedge *snd_scale'* = **option_case** 0 **I** *ws*

else

(* Otherwise, turn off scaling *)

rcv_scale' = 0 \wedge *snd_scale'* = 0) \wedge

(* Constrain the receive window for the new connection – this is advertised in the SYN,ACK reply. No scaling is performed here as scaling is not applied to segments containing a valid SYN since the support for window scaling has not been fully negotiated yet! *)

rcv_window \in {*n* | *n* \geq 0 \wedge
n \leq TCP_MAXWIN \wedge
n \leq *sf.n*(SO_RCVBUF)} \wedge

(* Time the SYN,ACK reply segment. This is a new connection thus no previous timers can be running. *)

(**let** *t_rttseg'* = \uparrow (ticks_of *h.ticks*, *cb.snd_nxt*) **in**

(* Initial sequence number of SYN,ACK reply segment is unconstrained. *)

iss \in {*n* | **T**} \wedge

(* The ack value in the reply segment must acknowledge the remote host's initial SYN. *)

let *ack'* = *seq* + 1 **in**

(* Update the new connection's control block in light of above. *)

cb' = *cb* \llbracket

tt_keep := \uparrow ((())_{slow_timer} TCPTV_KEEP_IDLE);

tt_rexmt := start_tt_rexmt *h.arch* 0 **F** *cb.t_rttinf*;

iss := *iss*;

irs := *seq*;

rcv_wnd := *rcv_window*;

tf_rxwin0sent := (*rcv_window* = 0);

rcv_adv := *ack'* + *rcv_window*;

rcv_nxt := *ack'*;

snd_una := *iss*;

snd_max := *iss* + 1; (* SYN consumes one-byte of sequence space *)

snd_nxt := *iss* + 1; (* SYN consumes one-byte of sequence space *)

snd_cwnd := *snd_cwnd'*;

rcv_up := *seq* + 1; (* Pull along with left edge of unused window *)

t_maxseg := *t_maxseg'*; (* The negotiated mss, with options removed *)

t_advms := *advms'*; (* Remember the mss advertised (if any) by this socket in case the SYN segment is retransmitted *)

rcv_scale := *rcv_scale'*;

snd_scale := *snd_scale'*;

tf_doing_ws := *tf_doing_ws'*;

ts_recent := **case** *ts* **of**

```

* → cb.ts_recent ||
↑(ts_val, ts_ecr) → (ts_val)kern_timer dtsinvalTIMEWINDOW;
last_ack_sent := ack';
t_rttseg := t_rttseg';
tf_req_tstamp := tf_doing_tstamp';
tf_doing_tstamp := tf_doing_tstamp'
⌋) ∧

```

(* Construct the SYN,ACK segment using the values stored in the updated control block for the new connection. See `make_syn_ack_segment` (p107). *)

choose $seg' :: \text{make_syn_ack_segment } cb'(i_1, i_2, p_1, p_2)(\text{ticks_of } h.\text{ticks}).$

(* Add the SYN,ACK reply segment to the host's output queue, ignoring failure. Constrain the new connection's initial control block cb to have just the right values in case queueing of the segment fails (perhaps due to a routing failure) and some control block state has to be rolled back. See `rollback_tcp_output` (p117) and `enqueue_or_fail` (p118) for more detail. *)

```

enqueue_or_fail T h.arch h.rttab h.ifds[TCP seg'] oq
(cb
  { snd_next := iss; (* If queueing fails, need to retransmit the SYN *)
    snd_max := iss; (* If queueing fails, need to retransmit the SYN *)
    t_maxseg := t_maxseg';
    last_ack_sent := tcp_seq_foreign 0w;
    rcv_adv := tcp_seq_foreign 0w
  })cb'(cb'', oq')

```

Model details

During TCP connection establishment, BSD uses syn-caches and syn-buckets to protect against some types of denial-of-service attack. These techniques delay the memory allocation for a socket's data structures until connection establishment is complete. They are not modelled directly in this specification, which instead favours the use of the full socket structure for clarity. The behaviour is observationally equivalent provided correct bounds are applied to the lengths of the incoming connection queues.

When a socket completes connection establishment, i.e., enters the ESTABLISHED state, BSD updates the socket's control block t_maxseg field to the minimum of the maximum segment size it advertised in the emitted SYN,ACK segment and that received in the SYN segment from the remote end. This update is later than perhaps it need be. This model updates the t_maxseg at the moment both the maximum segment values are known. As a consequence the initial maximum segment value advertised by the host must be stored just in case the SYN,ACK segment need be retransmitted.

Variations

| | |
|---------|--|
| FreeBSD | On FreeBSD, the <code>listen()</code> socket call can be called on a TCP socket in any state, thus it is possible for a listening TCP socket to have a peer address, i.e., is_2 and ps_2 pair, specified. This in turn affects the behaviour of connection establishment because an incoming <i>SYN</i> segment only matches this type of listening socket if its address quad matches the socket's entire address quad, heavily restricting the usefulness of such a socket. Such a restrictive peer address binding is permitted by the model for FreeBSD only. |
|---------|--|

deliver_in_1b **tcp: network nonurgent** For a listening socket, receive and drop a bad datagram and either generate a RST segment or ignore it. Drop the incoming segment if the socket's queue of

incomplete connections is full.

$$\begin{array}{l}
 h \langle \langle socks := socks \oplus [(sid, sock)]; \\
 iq := iq; \\
 oq := oq; \\
 bndlm := bndlm \rangle \rangle \xrightarrow{\tau} h \langle \langle socks := socks \oplus [(sid, sock)]; \\
 iq := iq'; \\
 oq := oq'; \\
 bndlm := bndlm' \rangle \rangle
 \end{array}$$

(* **Summary:** A host h with listening socket $sock$ referenced by index sid receives a segment seg addressed to socket $sock$. The segment either contains an invalid combination of the SYN and ACK flags, is a forged segment trying to force the listening socket $sock$ to connect to itself, or the new incomplete connection can not be added to the queue of incomplete connections because the completed connections queue is full. The segment is dropped. If the segment had the ACK flag set and not SYN , a RST segment is generated and added to the host's output queue oq for transmission. *)

(* Take TCP segment seg from the head of the host's input queue *)
 $dequeue_iq(iq, iq', \uparrow(\text{TCP } seg)) \wedge$

(* The segment must be of an acceptable form *)

(* Note: some segment fields are ignored during TCP connection establishment and as such may contain arbitrary values. These are equal to the identifiers postfixed with $_discard$ below, which are otherwise unconstrained. *)

$(\exists seq_discard \ ack_discard \ URG_discard \ PSH_discard \ FIN_discard$
 $win_discard \ ws_discard \ urp_discard \ mss_discard \ ts_discard \ data_discard.$

$$\begin{array}{l}
 seg = \langle \langle \\
 \quad is_1 := \uparrow i_2; \\
 \quad is_2 := \uparrow i_1; \\
 \quad ps_1 := \uparrow p_2; \\
 \quad ps_2 := \uparrow p_1; \\
 \quad seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign); \\
 \quad ack := tcp_seq_flip_sense(ack_discard : tcp_seq_local); \\
 \quad URG := URG_discard; \\
 \quad ACK := ACK; (* might be set in a bad SYN segment *) \\
 \quad PSH := PSH_discard; \\
 \quad RST := \mathbf{F}; (* SYN segments never have RST set *) \\
 \quad SYN := SYN; (* might not be set in a bad segment to a listening socket *) \\
 \quad FIN := FIN_discard; \\
 \quad win := win_discard; \\
 \quad ws := ws_discard; \\
 \quad urp := urp_discard; \\
 \quad mss := mss_discard; \\
 \quad ts := ts_discard; \\
 \quad data := data_discard \\
 \rangle \rangle \\
) \wedge
 \end{array}$$

(* Segment is addressed to an IP address belonging to one of the interfaces of host h and is not a link-layer multicast or IP-layer broadcast address *)

$i_1 \in local_lips \ h.ifds \wedge$
 $\neg(is_broadormulticast \ h.ifds \ i_1) \wedge (* \text{very unlikely, since } i_1 \in local_lips \ h.ifds *)$
 $\neg(is_broadormulticast \ h.ifds \ i_2) \wedge$

(* Find the socket $sock$ that has the best match for the address quad in segment seg , see $tcp_socket_best_match$ (p86). Socket $sock$ must have a form matching the patten $SOCK(\dots)$ *)

$tcp_socket_best_match(socks \setminus sid)(sid, sock) seg \ h.arch \wedge$
 $sock = SOCK(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
 $\quad TCP_Sock(LISTEN, cb, \uparrow lis, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

(* If socket $sock$ has a local IP address specified it should be the same as the destination IP address of segment seg . *)
(case is_1 **of** $\uparrow iI' \rightarrow iI' = i_1 \parallel * \rightarrow \mathbf{T}$ **)** \wedge

(* A BSD socket in the LISTEN state may have its peer's IP address is_2 and port ps_2 set because `listen()` can be called from any TCP state. On other architectures they are both constrained to *. *)

$((is_2 = * \wedge ps_2 = *) \vee$
 $(bsd_arch \ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

(* Check that either: (a) the *SYN*, *ACK* flag combination is bad, or (b) the socket is illegally connecting to itself (Note: it is not possible to perform a self-connect once a socket is in the LISTEN state by using the sockets interface alone – it can only be achieved by a forged incoming segment. It is possible for a TCP socket to connect to itself but this is achieved through a sequence of socket calls that avoids entering the LISTEN state), or (c) the new incomplete connection can not be added to the incomplete connections queue because the queue of complete connections is full. *)

$(ACK \vee$
 $(\neg SYN \wedge \neg ACK) \vee$
 $(SYN \wedge \neg ACK \wedge i_1 = i_2 \wedge p_1 = p_2) \vee$
 $accept_incoming_q0 \ lis \ \mathbf{F}$
 $) \wedge$

(* If an ACK with no SYN has been received send a RST segment, else just silently drop everything else. See `dropwithreset` (p120). *)

(if $\neg SYN \wedge ACK$ **then**
 $dropwithreset \ seg \ h.ifds(ticks_of \ h.ticks) BANDLIM_RST_OPENPORT \ bndlm \ bndlm' \ outsegs$
else
 $outsegs = [] \wedge bndlm' = bndlm) \wedge$

(* Add the RST segment (if any) to the host's output queue, ignoring failure. See `enqueue_and_ignore_fail` (p118). *)

$enqueue_and_ignore_fail \ h.arch \ h.rttab \ h.ifds \ outsegs \ oq \ oq'$

deliver_in_2 **tcp: network nonurgent** Completion of active open (in SYN_SENT receive SYN,ACK and send ACK) or simultaneous open (in SYN_SENT receive SYN and send SYN,ACK)

$h \langle socks := socks \oplus$
 $[(sid, SOCK(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es,$
 $cantsndmore, cantrcvmore, TCP_PROTO \ tcp_sock))];$

$iq := iq;$
 $oq := oq \rangle$

$\xrightarrow{\tau} h \langle socks := socks \oplus$
 $[(sid, SOCK(\uparrow fid, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es,$
 $cantsndmore, cantrcvmore',$
 $TCP_Sock(st', cb'', *, [], *, rcvq', rcvurp', iobc'))];$

$iq := iq';$
 $oq := oq' \rangle$

$tcp_sock = TCP_Sock0(SYN_SENT, cb, *, [], *, [], *, NO_OOBDATA) \wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
 $dequeue_iq(iq, iq', \uparrow(TCP \ seg)) \wedge$

$(\exists win_ws_urp_mss_PSH_discard.$

$win = \mathbf{w2n} \ win_ \wedge$

$ws = \mathbf{option_map \ ord} \ ws_ \wedge$

$urp = \mathbf{w2n} \ urp_ \wedge$

$mss = \mathbf{option_map \ w2n} \ mss_ \wedge$

$seg = \langle$
 $is_1 := \uparrow i_2;$
 $is_2 := \uparrow i_1;$
 $ps_1 := \uparrow p_2;$
 $ps_2 := \uparrow p_1;$

```

seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);
ack := tcp_seq_flip_sense(ack : tcp_seq_local);
URG := URG;
ACK := ACK;
PSH := PSH_discard;
RST := F;
SYN := T;
FIN := FIN;
win := win_;
ws := ws_;
urp := urp_;
mss := mss_;
ts := ts;
data := data
])  $\wedge$ 

```

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly *)

(* The ACK must be acceptable, else send RST. Typically (no data on active open), this is the same as $ack = iss + 1$ *)

```
(ACK  $\implies$  (cb.iss < ack  $\wedge$  ack  $\leq$  cb.snd_max))  $\wedge$ 
```

(* resolve negotiated window scaling *)

```
(case (cb.request_r_scale, ws) of
  ( $\uparrow$  rs,  $\uparrow$  ss)  $\rightarrow$  rcv_scale' = rs  $\wedge$ 
    snd_scale' = ss  $\wedge$ 
    tf_doing_ws' = T ||
  _15432  $\rightarrow$  rcv_scale' = 0  $\wedge$ 
    snd_scale' = 0  $\wedge$ 
    tf_doing_ws' = F)  $\wedge$ 
```

(* resolve negotiated timestamping *)

```
tf_rcvd_tstamp' = is_some ts  $\wedge$ 
tf_doing_tstamp' = (tf_rcvd_tstamp'  $\wedge$  cb.tf_req_tstamp)  $\wedge$ 
```

(* Note that for test generation at present we clear the route metric cache so this will always be NONE. BSD reads from the routing cache if there is an entry, otherwise passes NONE here. *)

```
bw_delay_product_for_rt = *  $\wedge$ 
```

```
let ourmss = (case cb.t_advmss of
  *  $\rightarrow$  cb.t_maxseg (* we did not advertise an MSS, so use the default value *)
  ||  $\uparrow$  v  $\rightarrow$  v) in
```

```
((rcvbufsize', sndbufsize', t_maxseg'', snd_cwnd') =
if mss  $\neq$  *  $\vee$   $\neg$ bsd_arch h.arch then
  calculate_buf_sizes ourmss mss bw_delay_product_for_rt
    (is_localnet h.ifds i2)(sf.n(SO_RCVBUF))
    (sf.n(SO_SNDBUF))tf_doing_tstamp' h.arch
else
  (* Note that since tcp_mss() is not called snd_cwnd remains at its initial (stupidly high) value. *)
  (sf.n(SO_RCVBUF), sf.n(SO_SNDBUF), cb.t_maxseg, cb.snd_cwnd)
)  $\wedge$ 
```

```
sf' = sf [ $\ll$  n := funupd_list sf.n[(SO_RCVBUF, rcvbufsize');
(SO_SNDBUF, sndbufsize')]]  $\wedge$ 
```

```
rcv_window = calculate_bsd_rcv_wnd sf' tcp_sock ∧
```

```
let (t_softerror', t_rttseg', t_rttinf', tt_rexmt')
```

```
= (if ACK then
```

```
  (* completion of active open. Conditions originally copied verbatim from deliver_in_3. *)
```

```
  (* update RTT estimators from timestamp or roundtrip time *)
```

```
  let emission_time = case ts of
```

```
    ↑(ts_val, ts_ecr) → ↑(ts_ecr - 1)
```

```
  || * →
```

```
  (case cb.t_rttseg of
```

```
    ↑(ts0, seq0) → if ack > seq0
```

```
      then ↑ ts0
```

```
      else *
```

```
  || * → *) in
```

```
  (* clear soft error, cancel timer, and update estimators if we successfully timed a segment round-trip *)
```

```
  let (t_softerror', t_rttseg', t_rttinf')
```

```
    = if is_some emission_time then
```

```
      (*,
```

```
      *,
```

```
      update_rtt(real_of_int(ticks_of h.ticks - the emission_time)/HZ)
```

```
      cb.t_rttinf)
```

```
    else
```

```
      (cb.t_softerror,
```

```
      cb.t_rttseg,
```

```
      cb.t_rttinf) in
```

```
  (* mess with retransmit timer if appropriate *)
```

```
  let tt_rexmt' =
```

```
  (if ack = cb.snd_max then
```

```
    (* if acked everything, stop *)
```

```
    *
```

```
    (* needoutput = 1 - see below *)
```

```
  else if mode_of cb.tt_rexmt = ↑ REXMTSYN then
```

```
    (* if partial ack, restart from current backoff value, which is always zero because of the above updates
```

```
    to the RTT estimators and shift value. *)
```

```
    start_tt_rexmtsyn h.arch 0 T t_rttinf')
```

```
  else if mode_of cb.tt_rexmt ∈ {*, ↑ REXMT} then
```

```
    (* ditto *)
```

```
    start_tt_rexmt h.arch 0 T t_rttinf')
```

```
  else if emission_time ≠ * then
```

```
    case cb.tt_rexmt of
```

```
      (* bizarre but true. tcp_input.c:1766 says c.f. Phil Karn's retransmit algorithm *)
```

```
      * → *
```

```
      || ↑(((mode, shift))d) → ↑(((mode, 0))d)
```

```
    else
```

```
      (* do nothing *)
```

```
      cb.tt_rexmt
```

```
  ) in
```

```
  (t_softerror',
```

```
  t_rttseg',
```

```
  t_rttinf',
```

```
  tt_rexmt')
```

```
else
```

```
  (* simultaneous open *)
```

```
(cb.t_softerror,
```



```

    cb.t_rttseg,
    cb.t_rttinf,
    start_tt_rexmt h.arch 0 T cb.t_rttinf) (* reset rexmt timer *)
) in

```

(* urgent pointer processing. See *deliver_in_3* for discussion (these conditions are originally copied verbatim from there). *)

```
(∃iobc rcvurp.
```

```
iobc = NO_OOBDATA ∧ (* we know the initial state has no OOB data *)
```

```
rcvurp = * ∧
```

```
(if URG ∧
```

```
    urp > 0 ∧
```

```
    urp + 0 ≤ SB_MAX
```

```
then
```

```
    (if seq + urp > cb.rcv_up then
```

```
        rcv_up' = seq + 1 + urp ∧
```

```
        rcvurp' = ↑(0 + num(seq + urp - cb.rcv_nxt))
```

```
    else
```

```
        rcv_up' = cb.rcv_nxt ∧ (* pull along with window *)
```

```
        rcvurp' = rcvurp) ∧
```

```
    (if urp ≤ length data ∧ sf.b(SO_OOBINLINE) = F then
```

```
        iobc' = OOBDATA(EL(urp - 1) data) ∧
```

```
        data_deoobed = (TAKE(urp - 1) data) @ (DROP urp data)
```

```
    else
```

```
        iobc' = (if seq + urp > cb.rcv_up then NO_OOBDATA else iobc) ∧
```

```
        data_deoobed = data)
```

```
else
```

```
    rcv_up' = seq + 1 ∧
```

```
    rcvurp' = rcvurp ∧
```

```
    iobc' = iobc ∧
```

```
    data_deoobed = data)
```

```
) ∧
```

(* data processing is much simpler here than in *deliver_in_3* because we know we will only ever receive the one *SYN*, *ACK* datagram (duplicates will be rejected, and there's only one datagram and so cannot be reordered). *)

```
data' = TAKE rcv_window data_deoobed ∧
```

```
FIN' = (if data' = data_deoobed then FIN else F) ∧
```

```
rcvq' = data' ∧ (* because rcvq is empty initially *)
```

```
rcv_nxt' = seq + 1 + length data' + (if FIN' then 1 else 0) ∧
```

```
rcv_wnd' = rcv_window - length data' ∧
```

```
cb' = cb {
```

```
    tt_rexmt := tt_rexmt';
```

```
    (* not persist, because we do not have any data to send *)
```

```
    t_idletime := stopwatch_zero; (* just received a segment *)
```

```
    tt_keep := ↑((( ))_slow_timer TCPTV_KEEP_IDLE);
```

```
    tt_conn_est := *;
```

```
    tt_delack := *;
```

```
    snd_una := ack onlywhen ACK; (* = cb.iss + 1, or +2 if full ack of SYN,FIN *)
```

```
    snd_nxt := ack onlywhen(ACK ∧ cantsndmore); (* prepare for possible outbound FIN *)
```

```
    snd_max := ack onlywhen(ACK ∧ cantsndmore ∧ ack > cb.snd_max);
```

```
    (* we doubt snd_max can ever increase here, but put this in for safety *)
```

```

snd_wl1 := if ACK then seq + 1 else seq; (* must update window. c.f. TCPv2p951, TCPv2p981f,
                                         and tcp_input.c:1824 *)
snd_wl2 := ack onlywhen ACK;
snd_wnd := win << snd_scale';
snd_cwnd := if ACK ∧ ack > cb.iss + 1 then
  (* BSD clamps snd_cwnd to the maximum window size (65535), but only if we received an ack for data
  other than the initial SYN. See tcp_input.c:1791 *)
  min(snd_cwnd')(TCP_MAXWIN << snd_scale')
else
  snd_cwnd';
rcv_scale := rcv_scale';
snd_scale := snd_scale';
tf_doing_ws := tf_doing_ws';
irs := seq;
rcv_nxt := rcv_nxt';
rcv_wnd := rcv_wnd';
tf_rwin0sent := (rcv_wnd' = 0);
rcv_adv := rcv_nxt' + (rcv_wnd' >> rcv_scale') << rcv_scale';
rcv_up := rcv_up';
t_maxseg := t_maxseg'';
ts_recent := case ts of
  (* record irrespective of whether we negotiated to do this or not, like BSD *)
  * → cb.ts_recent ||
  ↑(ts_val, ts_ecr) → (ts_val)kern_timerTIMEWINDOW dtsinval;
  (* timestamp will become invalid in 24 days *)
last_ack_sent := rcv_nxt';
t_softerror := t_softerror';
t_rttseg := t_rttseg';
t_rttinf := t_rttinf';
tf_req_tstmp := tf_doing_tstmp';
tf_doing_tstmp := tf_doing_tstmp'
  ] ∧

```

(* now generate seg' , unless we're delaying the ACK *)

```

(choose  $seg'$  :: (if ACK then
  (* completion of active open *)
  make_ack_segment  $cb'$ (cantsndmore ∧ ack < cb.iss + 2)( $i_1, i_2, p_1, p_2$ )(ticks_of  $h.ticks$ )
else
  (* simultaneous open *)

```

```

let  $cb'''$  =
  (if ((linux_arch  $h.arch$ ) ∧  $cb.tf\_req\_tstmp$ ) then
     $cb'$  {  $tf\_req\_tstmp := \mathbf{T}$ ;
           $tf\_doing\_tstmp := \mathbf{T}$  }
  else
     $cb'$  in

  (if bsd_arch  $h.arch$  then
    make_ack_segment  $cb'''$  F( $i_1, i_2, p_1, p_2$ )(ticks_of  $h.ticks$ )
  else
    make_syn_ack_segment  $cb'''$ ( $i_1, i_2, p_1, p_2$ )(ticks_of  $h.ticks$ )).

```

(* Add the segment to the host's output queue. See enqueue_or_fail (p118). *)

```

enqueue_or_fail T h.arch h.rtttab h.ifds[TCP seg']oq
  (cb [t_rttinf := cb'.t_rttinf;
      t_maxseg := t_maxseg'';
      snd_nxt := cb.snd_nxt;
      tt_delack := cb.tt_delack;
      last_ack_sent := cb.last_ack_sent;
      rcv_adv := cb.rcv_adv
    ]) cb'(cb'', oq')
)  $\wedge$ 

```

(* Note that we change state even if enqueueing or routing returned an error, trusting to retransmit to solve our problem. *)

```

(if ACK then
  (* completion of active open *)
  (if  $\neg$ FIN' then
    (cantrcvmore' = cantrcvmore  $\wedge$ 
      st' =
      (if cantsndmore = F then
        ESTABLISHED
      else if cb.snd_max > cb.iss + 1  $\wedge$  ack  $\geq$  cb.snd_max then (* our FIN is ACKed *)
        FIN_WAIT_2
      else
        FIN_WAIT_1)) (* we were trying to send a FIN from SYN_SENT, so move straight to
        FIN_WAIT_2. Definitely the case with BSD; should also be true for other archs. *)
    else
      (cantrcvmore' = T  $\wedge$ 
        st' =
        (if cantsndmore = F then
          CLOSE_WAIT
        else
          LAST_ACK))) (* we were trying to send a FIN from SYN_SENT and also receive a FIN, so we
          move straight into LAST_ACK. *)
  else
    (* simultaneous open *)
    (if  $\neg$ FIN' then
      (st' = SYN_RECEIVED  $\wedge$ 
        cantrcvmore' = cantrcvmore)
      else
        (st' = CLOSE_WAIT  $\wedge$  (* yes, really! (in BSD) even though we've not yet had our initial SYN acknowl-
          edged! See tcp_input.c:2065 +/-2000 *)
          cantrcvmore' = T))
    )
)

```

deliver_in_2a **tcp: network nonurgent** Receive bad or boring datagram and *RST* or ignore for *SYN_SENT* socket

```

h [socks := socks  $\oplus$     $\xrightarrow{\tau}$    h [socks := socks  $\oplus$ 
  [(sid, sock);          [(sid, sock');
iq := iq;                iq := iq';
oq := oq;                oq := oq';
bndlm := bndlm}]         bndlm := bndlm'}]

```

(* **Summary:** For a *SYN_SENT* socket unacceptable acks get *RST*ed; boring but otherwise OK segments are ignored. *)

$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \text{TCP_Sock}(\text{SYN_SENT}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

(* Take TCP segment seg from the head of the host's input queue *)
 $\text{dequeue_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$

$(\exists seq_discard \text{ URG_discard } \text{PSH_discard } \text{FIN_discard}$
 $\text{win_discard } \text{ws_discard } \text{urp_discard } \text{mss_discard } \text{ts_discard } \text{data_discard}.$

$seg = \langle$
 $is_1 := \uparrow i_2;$
 $is_2 := \uparrow i_1;$
 $ps_1 := \uparrow p_2;$
 $ps_2 := \uparrow p_1;$
 $seq := \text{tcp_seq_flip_sense}(seq_discard : \text{tcp_seq_foreign});$
 $ack := \text{tcp_seq_flip_sense}(ack : \text{tcp_seq_local});$
 $\text{URG} := \text{URG_discard};$
 $\text{ACK} := \text{ACK};$
 $\text{PSH} := \text{PSH_discard};$
 $\text{RST} := \mathbf{F};$
 $\text{SYN} := \text{SYN};$
 $\text{FIN} := \text{FIN_discard};$
 $\text{win} := \text{win_discard};$
 $\text{ws} := \text{ws_discard};$
 $\text{urp} := \text{urp_discard};$
 $\text{mss} := \text{mss_discard};$
 $\text{ts} := \text{ts_discard};$
 $\text{data} := \text{data_discard}$
 \rangle
 $\rangle \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

$((\text{ACK} \wedge \neg(cb.iss < ack \wedge ack \leq cb.snd_max)) \vee$
 $(\neg\text{SYN} \wedge (\neg\text{ACK} \vee (\text{ACK} \wedge cb.iss < ack \wedge ack \leq cb.snd_max)))) \wedge$

(if $\text{ACK} \wedge \neg(cb.iss < ack \wedge ack \leq cb.snd_max)$ **then**
 $\text{dropwithreset } seg \ h.ifds(\text{ticks_of } h.ticks)\text{BANDLIM_UNLIMITED } bndlm \ bndlm' \ \text{outsegs}$
else if $\neg\text{SYN} \wedge (\neg\text{ACK} \vee (\text{ACK} \wedge cb.iss < ack \wedge ack \leq cb.snd_max))$ **then**
 $\text{outsegs} = [] \wedge bndlm' = bndlm$
else
 \mathbf{F}) \wedge

let $tcp_sock = \text{tcp_sock_of } sock$ **in**
 (* BSD rcv_wnd bug: the receive window updated code in tcp_input gets executed *before* the segment is processed, so even for bad segments, it gets updated. *)

let $rcv_window = \text{calculate_bsd_rcv_wnd } sf \ tcp_sock$ **in**
 $sock' = sock \langle$ $pr := \text{TCP_PROTO}(tcp_sock$
 \langle $cb := tcp_sock.cb$
 \langle $rcv_wnd := \text{if } \text{bsd_arch } h.arch \ \text{then } rcv_window \ \text{else } tcp_sock.cb.rcv_wnd;$
 $\text{rcv_adv} := \text{if } \text{bsd_arch } h.arch \ \text{then } tcp_sock.cb.rcv_nxt + rcv_window$
 $\text{else } tcp_sock.cb.rcv_adv;$
 $t_idletime := \text{stopwatch_zero};$
 $tt_keep := \uparrow(((\))_{\text{slow_timer}} \ \text{TCPTV_KEEP_IDLE})$
 $\rangle\rangle\rangle \wedge$

$\text{enqueue_and_ignore_fail } h.arch \ h.rttab \ h.ifds \ \text{outsegs} \ oq \ oq'$

(* For a socket in the SYN_RECEIVED state check that the ACK is valid (the acknowledge value *ack* is not outside the range of sequence numbers that have been transmitted to the remote socket) and that the segment is not a LAND DoS attack (the segment's sequence number is not smaller than the remote socket's (the receiver from this socket's perspective) initial sequence number) *)
 $\neg(tcp_sock.st = SYN_RECEIVED \wedge$
 $((ACK \wedge (ack \leq tcp_sock.cb.snd_una \vee ack > tcp_sock.cb.snd_max)) \vee$
 $seq < tcp_sock.cb.irs)) \wedge$

(* If socket *sock* has previously emitted a *FIN* segment check that a thread is still associated with the socket, i.e. check that the socket still has a valid file identifier *fid* $\neq *$. If not, and the segment contains new data, the segment should not be processed by this rule as there is no thread to read the data from the socket after processing. Query: how does this *st* condition relate to *wesentafin* below? *)
 $\neg(tcp_sock.st \in \{FIN_WAIT_1; CLOSING; LAST_ACK; FIN_WAIT_2; TIME_WAIT\} \wedge$
 $sock.fid = * \wedge$
 $seq + \mathbf{length} \ data > tcp_sock.cb.rcv_nxt) \wedge$

(* A *SYN* should be received only in the SYN_RECEIVED state. *)
 $(SYN \implies tcp_sock.st = SYN_RECEIVED) \wedge$

(* Socket *sock* has previously sent a *FIN* segment iff *snd_max* is strictly greater than the sequence number of the byte after the last byte in the send queue *sndq*. *)
let *wesentafin* = $tcp_sock.cb.snd_max > tcp_sock.cb.snd_una + \mathbf{length} \ tcp_sock.sndq$ **in**

(* If the socket *sock* has previously sent a *FIN* segment it has been acknowledged by segment *seg* if the segment has the *ACK* flag set and an acknowledgment number $ack \geq cb.snd_max$. *)
let *ourfinisacked* = $(wesentafin \wedge ACK \wedge ack \geq tcp_sock.cb.snd_max)$ **in**

(* Process the segment and return an updated socket state *)
 (* The segment processing is performed by the four relations below, i.e., *di3_topstuff*, *di3_ackstuff*, *di3_datastuff* and *di3_ststuff*. Each of these relates a socket and bandwidth limiter state before the segment is processed to a tuple containing an updated socket, new bandwidth limiter state, a list of zero or more segments to output and a continue flag. The aim is to model the progression of the segment through *tcp_input()*. When the continue flag is **T** segment processing should continue. The infix function *andThen* applies the function on its left hand side and only continues with the function on its right hand side if the left hand function's continue flag is **T**. For a further explanation of this relational monad behaviour see *aux_relmonad* (p??). *)

let *topstuff* =

(* Initial processing of the segment: PAWS (protection against wrap sequence numbers); ensure segment is not entirely off the right hand edge of the window; timer updates, etc. For further information see *di3_topstuff* (p294).*)
 $di3_topstuff \ seg \ h.arch \ h.rttab \ h.ifds(ticks_of \ h.ticks)$

and *ackstuff* =

(* Process the segment's acknowledgement number and do congestion control. See *di3_ackstuff* (p298).*)
 $di3_ackstuff \ tcp_sock \ seg \ ourfinisacked \ h.arch \ h.rttab \ h.ifds(ticks_of \ h.ticks)$

and *datastuff* *theststuff* =

(* Extract and reassemble data (including urgent data). See *di3_datastuff* (p304). *)
 $di3_datastuff \ thestuff \ tcp_sock \ seg \ ourfinisacked \ h.arch$

and *ststuff* *FIN_reass* =

(* Possibly change the socket's state (especially on receipt of a valid *FIN*). See *di3_ststuff* (p305). *)
 $di3_ststuff \ FIN_reass \ ourfinisacked \ ack$

in

$(topstuff \ \mathbf{andThen}$

$\ \ \ ackstuff \ \ \mathbf{andThen}$

$\ \ \ \ datastuff \ ststuff)$

$(sock, bndlm)$ (* state before *)

$((sock', bndlm', outsegs), continue') \wedge$ (* state after *)

$sock'.pr = TCP_PROTO(tcp_sock') \wedge$

(* If socket *sock* was initially in the SYN_RECEIVED state and after processing *seg* is in the ESTABLISHED state (or if the segment contained a *FIN* and the socket is in one of the FIN_WAIT_1, FIN_WAIT_2 or CLOSE_WAIT states), the socket is probably on some other socket's incomplete connections queue and *seg* is the final segment in a passive open. If it is on some other socket's incomplete connections queue the other socket is updated to move the newly connected socket's reference from the incomplete to the complete connections queue (unless the complete connection queue is full, in which case the new connection is dropped and all references to it are removed). If not, *seg* is the final segment in a simultaneous open in which case no other sockets are updated. The auxiliary function `di3_socks_update` (p308) does all the hard work, updating the relevant sockets in the finite map *socks* to yield *socks'*. *)

```
(if tcp_sock.st = SYN_RECEIVED ^
  tcp_sock'.st ∈ {ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSE_WAIT} then
  di3_socks_update sid(socks ⊕ (sid, sock'))socks'
else
  (* If the socket was not initially in the SYN_RECEIVED state, i.e. seg was processed by an already connected
  socket, ensure the updated socket is in the final finite maps of sockets. *)
  socks' = socks ⊕ (sid, sock')) ^
```

(* Queue any segments for output on the host's output queue. In the common case there are no segments to be output as output is handled by *deliver_out_1* etc. The exception is that `di3_ackstuff` (and its auxiliaries) require an immediate *ACK* segment to be emitted under certain congestion control conditions. See `di3_ackstuff` (p298) and `di3_newackstuff` (p295) for further details. *)

```
enqueue_oq_list_qinfo(oq, outsegs, oq')
```

– *deliver_in_3* initial checks :

```
di3_topstuff seg arch rttab ifds ticks =
```

(* monadic state accessor: *sock* is the socket processing the segment, as determined by *deliver_in_3* *)

```
(get_sock λsock.
```

(* Pull out the TCP protocol and control blocks *)

```
let tcp_sock = tcp_sock_of sock in
```

```
let cb = tcp_sock.cb in
```

(* If the segment has the *SYN* flag set, increment the sequence number so that it is the sequence number of the first byte of data in the segment *)

```
let seq = tcp_seq_flip_sense seg.seq + (if seg.SYN then 1 else 0) in
```

(* The sequence number of the byte logically after the last byte of data in the segment *)

```
let rseq = seq + length seg.data in
```

```
let ts = seg.ts in
```

(* PAWS (Protection Against Wrapped Sequence numbers) check: If the segment contains a timestamp value that is strictly less than *ts_recent* then the segment is invalid and the PAWS check fails. The value *ts_recent* is the timestamp value of the most recent of the previous segments that was successfully processed, i.e., the last segment that *deliver_in_3* processed without dropping. *)

```
let paws_failed =
```

```
(∃ts_val ts_ecr ts_recent.
```

```
  ts = ↑(ts_val, ts_ecr) ^ (* segment's timestamp field is a pair *)
```

```
  timewindow_val_of cb.ts_recent = ↑ ts_recent ^ (* most recent timestamp recorded *)
```

```
  ts_val < ts_recent) in (* check the segment's timestamp is not old *)
```

(* If the segment lies entirely off the right-hand edge of *sock*'s receive window then it should be dropped, provided it is not a window probe. *)

```
let segment_off_right_hand_edge =
```

```
(let rcv_wnd' = calculate_bsd_rcv_wnd sock.sf tcp_sock in (* size of receive window *)
```

```
  (seq ≥ cb.rcv_nxt + rcv_wnd') ^ (* segment starts on or after the right hand edge *)
```

```
  (rseq > cb.rcv_nxt + rcv_wnd') ^ (* segment ends after the right hand edge *)
```

```
  (rcv_wnd' ≠ 0)) in (* The segment is not a window probe, i.e., rcv_wnd' is not zero *)
```

(* Drop the segment being processed if either the PAWS check or the "off right hand edge of window" checks fail *)

```
let drop_it = (paws_failed ∨ segment_off_right_hand_edge) in
```

(* The value *ts_recent* will be updated to hold the value of the segment's timestamp field if the segment is not dropped. Timestamps are invalidated after 24 days - this is ensured by the attached kernel timer *kern_timer dtsinval*. *)

```
let ts_recent' = (fst(the ts))TIMEWINDOWkern_timer dtsinval in
```

(* Reset the socket's idle timer and keepalive timer to start counting from zero as activity is taking place on the socket: a segment is being processed. If the FIN_WAIT_2 timer is enabled this may be reset upon processing this segment. See *update_idle* (p119) for further details *)

```
let (t_idletime', tt_keep', tt_fin_wait_2') = update_idle tcp_sock in
```

(* Using the monadic state accessor *modify_cb* (p??), update the socket's control block with the new timer values and the most recent timestamp seen.

The *ts_recent* field is only updated if the segment currently being processed is not scheduled to be dropped, has a timestamp value set and is from a segment whose first byte of data has sequence number less than or equal to the last acknowledgement number sent in a segment to the remote end. The last condition (when coupled with the PAWS check above) ensures that *ts_recent* only increases monotonically and as is only updated by either a duplicate segment with a newer timestamp, or the next in-order segment expected by the receiving socket with a newer timestamp. It would be incorrect to record the newer timestamps of out-of-order segments because they would fail the PAWS check and get dropped

Note: if a reasonably continuous stream of segments is being received with increasing timestamp values and few data segments are sent in return such that acknowledgments are delayed, i.e., every other segment is acknowledged, then only the timestamp from every other segment is recorded by these conditions. This is still sufficient to protect against wrapped sequence numbers. *)

```
modify_cb(λcb'.cb' [ tt_keep := tt_keep';
                    tt_fin_wait_2 := tt_fin_wait_2';
                    t_idletime := t_idletime';
                    ts_recent := ts_recent' onlywhen
                    (¬drop_it ∧ is_some ts ∧ seq ≤ cb.last_ack_sent)
                    ]) andThen
```

if *drop_it* **then**

(* Decided to drop the segment. *mlift_dropafterack_or_fail* (p120) may decide to *RST* the connection depending upon the socket state. If so, the *RST* segment is retained on the monadic output segment list returned to *deliver_in_3* for queuing. *)

```
mlift_dropafterack_or_fail seg arch rttab ifds ticks andThen
```

(* After dropping, stop processing the segment. No need to waste time processing the segment any further *)

```
stop
```

```
else
```

(* Otherwise the segment is valid so allow processing to continue. *)

```
cont
```

```
)
```

– *deliver_in_3* **new** **ack** **processing**, **used** **in** *di3_ackstuff* :

```
di3_newackstuff tcp_sock_0 seg ourfinisacked arch rttab ifds ticks =
```

(* Pull some fields out of the segment *)

```
let ack = tcp_seq_flip_sense seg.ack in
```

```
let ts = seg.ts in
```

(* Get the socket's control block using the monadic state accessor *get_cb*. *)

```
(get_cb λcb'.
```

(if \neg TCP_DO_NEWRENO \vee *cb'.t_dupacks* < 3 **then**

(* If not doing NewReno-style Fast Retransmit or there have been fewer than 3 duplicate *ACKS* then clear the duplicate *ACK* counter. If there were more than 3 duplicate *ACKS* previously then the congestion window was inflated as per RFC2581 so retract it to *snd_ssthresh* *)

```
modify_cb(λcb'.cb' [ t_dupacks := 0;
```

```
                    snd_cwnd := (min cb'.snd_cwnd cb'.snd_ssthresh) (* retract the window safely *)
```



```

onlywhen( $cb'.t\_dupacks \geq 3$ )
else if TCP_DO_NEWRENO  $\wedge cb'.t\_dupacks \geq 3 \wedge ack < cb'.snd\_recover$  then
  (* The host supports NewReno-style Fast Recovery, the socket has received at least three duplicate ACKs previously and the new ACK does not complete the recovery process, i.e., there are further losses or network delays. The new ACK is a partial ACK per RFC2582. Perform a retransmit of the next unacknowledged segment and deflate the congestion window as per the RFC. *)
  modify_cb( $\lambda cb'.cb'$  [
    (* Clear the retransmit timer and round-trip time measurement timer. These will be started by tcp_output_really when the retransmit is actioned. *)
     $tt\_rext := *$ ;
     $t\_rttseg := *$ ;

    (* Segment to retransmit starts here *)
     $snd\_nxt := ack$ ;

    (* Allow one segment to be emitted *)
     $snd\_cwnd := cb'.t\_maxseg$ 
  ]) andThen

  (* Attempt to create a segment for output using the modified control block (this is a relational monad idiom) *)
  mlift_tcp_output_perhaps_or_fail  $ticks arch rttab ifds$  andThen

  (* Finally update the control block: *)
  modify_cb( $\lambda cb'.cb'$  [
    (* RFC2582 partial window deflation: deflate the congestion window by the amount of data freshly acknowledged and add back one maximum segment size *)
     $snd\_cwnd := \mathbf{num}(\mathbf{int\_of\_num} \ cb'.snd\_cwnd -$ 
       $(ack - cb'.snd\_una) + \mathbf{int\_of\_num} \ cb'.t\_maxseg);$ 
     $snd\_nxt := cb'.snd\_nxt)$  (* restore previous value *)
  ])

else if TCP_DO_NEWRENO  $\wedge cb'.t\_dupacks \geq 3 \wedge ack \geq cb'.snd\_recover$  then
  (* The host supports NewReno-style Fast Recovery, the socket has received at least three duplicate ACK segments and the new ACK acknowledges at least everything upto  $snd\_recover$ , completing the recovery process. *)

  modify_cb( $\lambda cb'.cb'$  [  $t\_dupacks := 0$ ; (* clear the duplicate ACK counter *)
    (* Open up the congestion window, being careful to avoid an RFC2582 Ch3.5 Pg6 "burst of data". *)
     $snd\_cwnd :=$ 
    if  $cb'.snd\_max - ack < \mathbf{int\_of\_num} \ cb'.snd\_ssthresh$  then
      (* If  $snd\_ssthresh$  is greater than the number of bytes of data still unacknowledged and presumed to be in-flight, set  $snd\_cwnd$  to be one segment larger than the total size of all the segments in flight. This is burst avoidance:  $tcp\_output$  is only able to send upto one further segment until some of the in flight data is acknowledged. *)
       $\mathbf{num}(cb'.snd\_max - ack + \mathbf{int\_of\_num} \ cb'.t\_maxseg)$ 
    else
      (* Otherwise, set  $snd\_cwnd$  to be  $snd\_ssthresh$ , forbidding any further segment output until some in flight data is acknowledged.*)
       $cb'.snd\_ssthresh$ 
    ])

  else  $assert\_failure$ "di3_newackstuff" (* impossible *)

) andThen

(* Check  $ack$  value is sensible, i.e., not greater than the highest sequence number transmitted so far *)
if  $ack > cb'.snd\_max$  then
  (* Drop the segment and possibly emit a RST segment *)
  mlift_dropafterack_or_fail  $seg arch rttab ifds ticks$  andThen
  stop

```

```

else (* continue processing *)
(* If the retransmit timer is set and the socket has done only one retransmit and it is still within the bad retransmit
timer window, then because this is an ACK of new data the retransmission was done in error. Flag this so that the
control block can be recovered from retransmission mode. This is known as a "bad retransmit". *)
let revert_rexmt = (mode_of cb'.tt_rexmt ∈ {↑ REXMT; ↑ REXMTSYN}) ∧
    shift_of cb'.tt_rexmt = 1 ∧ timewindow_open cb'.t_badrxtwin) in

(* Attempt to calculate a new round-trip time estimate *)
let emission_time = case (ts, cb'.t_rttseg) of
    (↑(ts_val, ts_ecr), -) →
        (* By using the segment's timestamp if it has one *)
        ↑(ts_ecr - 1)
    || (*, ↑(ts0, seq0)) →
        (* Or if not, by the control blocks round-trip timer, if it covers the segment(s) being
        acknowledged *)
        if ack > seq0 then ↑ ts0 else *
    || (*, *) →
        (* Otherwise, it is not possible to calculate a round-trip update *)
        * in

(* If a new round-trip time estimate was calculated above, update the round-trip information held by the socket's
control block *)
let t_rttinf' = case emission_time of
    ↑ t_rttinf → update_rtt(real_of_int(ticks - the emission_time)/HZ)
        cb'.t_rttinf
    || * → cb'.t_rttinf in

(* Update the retransmit timer *)
let tt_rexmt' =
(if ack = cb'.snd_max then
    * (* If all sent data has been acknowledged, disable the timer *)
else case mode_of cb'.tt_rexmt of
    * →
        (* If not set, set it as there is still unacknowledged data *)
        start_tt_rexmt arch 0 T t_rttinf'
    || ↑ REXMT →
        (* If set, reset it as a new acknowledgement segment has arrived *)
        start_tt_rexmt arch 0 T t_rttinf'
    || -444 →
        (* Otherwise, leave it alone. The timer will never be in REXMTSYN here and the only other case is PERSIST,
        in which case it should be left alone until such time as a window update is received *)
        cb'.tt_rexmt
) in

(* Update the send queue and window *)
let (snd_wnd', sndq') = (if ourfinisacked then
    (* If this socket has previously emitted a FIN segment and the FIN has now been
    ACKed, decrease snd_wnd by the length of the send queue and clear the send queue. *)
    (cb'.snd_wnd - length tcp_sock_0.sndq, [])
else
    (* Otherwise, reduce the send window by the amount of data acknowledged as it is now
    consuming space on the receiver's receive queue. Remove the acknowledged bytes from
    the send queue as they will never need to be retransmitted. *)
    (cb'.snd_wnd - num(ack - tcp_sock_0.cb.snd_una),
    DROP(num(ack - tcp_sock_0.cb.snd_una))tcp_sock_0.sndq)
) in

(* Update the control block *)
modify_cb(λcb.cb
⌈ (* If revert_rexmt (above) flags that a bad retransmission occurred, undo the congestion avoidance changes *)

```

```

    snd_cwnd := cb.snd_cwnd_prev onlywhen revert_rexmt;
    snd_ssthresh := cb.snd_ssthresh_prev onlywhen revert_rexmt;
    snd_nxt := cb'.snd_max onlywhen revert_rexmt;
    t_badrtwin := TIMEWINDOWCLOSED onlywhen revert_rexmt
  })andThen
  modify_cb(λcb.cb
  (
    (* Update the round-trip time estimates and retransmit timer *)
    t_rttinf := t_rttinf';
    tt_rexmt := tt_rexmt';

    (* If the ACK segment allowed us to successfully time a segment (and update the round-trip time estimates) then
    clear the soft error flag and clear the segment round-trip timer in order that it can be used on a future segment. *)
    t_softerror := * onlywhen is_some emission_time;
    t_rttseg := * onlywhen is_some emission_time;

    (* Update the congestion window by the algorithm in expand_cwnd (p99) only when not performing NewReno
    retransmission or the duplicate ACK counter is zero, i.e., expand the congestion window when this ACK is not a
    NewReno-style partial ACK and hence the connection has yet recovered *)
    snd_cwnd := expand_cwnd cb.snd_ssthresh tcp_sock_0.cb.t_maxseg
      (TCP_MAXWIN ≤ tcp_sock_0.cb.snd_scale)cb.snd_cwnd
      onlywhen(¬TCP_DO_NEWRENO ∨ cb'.t_dupacks = 0);
    snd_wnd := snd_wnd'; (* The updated send window *)
    snd_una := ack; (* Have had up to ack acknowledged *)
    snd_nxt := max ack cb.snd_nxt; (* Ensure invariant snd_nxt ≥ snd_una *)

    (* Reset the 2MSL timer if in the TIME_WAIT state as have received a valid ACK segment for the waiting socket *)
    tt_2msl := ↑((( ))_slow_timer(2*TCPTV_MSL))
      onlywhen(tcp_sock_0.st = TIME_WAIT)
  })andThen
  modify_tcp_sock(λs.s ( sndq := sndq' )) andThen (* The send queue update *)

(if tcp_sock_0.st = LAST_ACK ∧ ourfinisacked then
  (* If the socket's FIN has been acknowledged and the socket is in the LAST_ACK state, close the socket and stop
  processing this segment *)
  modify_sock(tcp_close arch) andThen
  stop
else if tcp_sock_0.st = TIME_WAIT ∧ ack > tcp_sock_0.cb.snd_una(* data acked past FIN *) then
  (* If the socket is in TIME_WAIT and this segment contains a new acknowledgement (that acknowledges past the
  FIN segment, drop it—it's invalid. Stop processing. *)
  mlift_dropafterack_or_fail seg arch rttab ifds ticks andThen
  stop
else
  (* Otherwise, flag that deliver_in_3 can continue processing the segment if need be *)
  cont)
)(* cb' *)

```

– *deliver_in_3* ACK processing :

```

di3_ackstuff tcp_sock_0 seg ourfinisacked arch rttab ifds ticks =
  (* Pull some fields out of the segment *)
let ack = tcp_seq_flip_sense seg.ack in
let seq = tcp_seq_flip_sense seg.seq in
let data = seg.data in

```

(* Pull out senders advertised window from the segment, applying the sender's scaling *)

let *win* = **w2n** *seg.win* << *tcp_sock_0.cb.snd_scale* **in**

(* Get the socket's control block using the monadic state accessor *get_cb*. Process the acknowledgement data in the segment, do some congestion control calculations and finally update the control blocks *)

(*get_cb* λ *cb*).

(* The segment is possibly a duplicate ack if it contains no data, does not contain a window update and the socket has unacknowledged data (the retransmit timer is still active). The no data condition is important: if this socket is sending little or no data at present and is waiting for some previous data to be acknowledged, but is receiving data filled segments from the other end, these may all contain the same acknowledgement number and trigger the retransmit logic erroneously. *)

let *has_data* = (*data* \neq [] \wedge

(*bsd_arch arch* \implies (*cb.rcv_nxt* < *seq* + **length** *data* \wedge *seq* < *cb.rcv_nxt* + *cb.rcv_wnd*))) **in**

let *maybe_dup_ack* = (\neg *has_data* \wedge *win* = *cb.snd_wnd* \wedge *mode_of cb.tt_rexmt* = \uparrow REXMT) **in**

if *ack* \leq *cb.snd_una* \wedge *maybe_dup_ack* **then**

(* Received a duplicate acknowledgement: it is an old acknowledgement (strictly less than *snd_una*) and it meets the duplicate acknowledgement conditions above. Do Fast Retransmit/Fast Recovery Congestion Control (RFC 2581 Ch3.2 Pg6) and NewReno-style Fast Recovery (RFC 2582, Ch3 Pg3), updating the control block variables and creating segments for transmission as appropriate. *)

let *t_dupacks'* = *cb.t_dupacks* + 1 **in**

if *t_dupacks'* < 3 **then**

(* Fewer than three duplicate acks received so far. Just increment the duplicate ack counter. We must continue processing, in case *FIN* is set. *)

modify_cb(λ *cb'.cb'* { *t_dupacks* := *t_dupacks'* }) **andThen**

cont

else if *t_dupacks'* > 3 \vee (*t_dupacks'* = 3 \wedge TCP_DO_NEWRENO \wedge *ack* < *cb.snd_recover*) **then**

(* If this is the 4th or higher duplicate ACK then Fast Retransmit/Fast Recovery congestion control is already in progress. Increase the congestion window by another maximum segment size (as the duplicate ACK indicates another out-of-order segment has been received by the other end and is no longer consuming network resource), increment the duplicate ACK counter, and attempt to output another segment. *)

(* If this is the 3rd duplicate ACK, the host supports NewReno extensions and *ack* is strictly less than the fast recovery "recovered" sequence number *snd_recover*, then the host is already doing NewReno-style fast recovery and has possibly falsely retransmitted a segment, the retransmitted segment has been lost or it has been delayed. Reset the duplicate ACK counter, increase the congestion window by a maximum segment size (for the same reason as before) and attempt to output another segment. NB: this will not cause a cycle to develop! The retransmission timer will eventually fire if recovery does not happen "fast". *)

modify_cb(λ *cb'.cb'* { *t_dupacks* := **if** *t_dupacks'* = 3 **then** 0 (* false retransmit, or further loss or delay *)

else *t_dupacks'*;

snd_cwnd := *cb.snd_cwnd* + *cb.t_maxseg* }) **andThen**

mlift_tcp_output_perhaps_or_fail ticks arch rttab ifds **andThen**

stop (* no need to process the segment any further *)

else if *t_dupacks'* = 3 \wedge \neg (TCP_DO_NEWRENO \wedge *ack* < *cb.snd_recover*) **then**

(* If this is the 3rd duplicate segment and if the host supports NewReno extensions, a NewReno-style Fast Retransmit is not already in progress, then do a Fast Retransmit *)

(* Update the control block before the retransmit to reflect which data requires retransmission *)

modify_cb(λ *cb'.cb'* { *t_dupacks* := *t_dupacks'*; (* increment the counter *)

(* Set to half the current flight size as per RFC2581/2582 *)

snd_ssthresh := **max** 2((**min** *cb.snd_wnd* *cb.snd_cwnd*) **div** 2

div *cb.t_maxseg*) * *cb.t_maxseg*;

(* If doing NewReno-style Fast Retransmit set to the highest sequence number transmitted so far *snd_max*. *)

snd_recover := *cb.snd_max* **onlywhen** TCP_DO_NEWRENO;

```

(* Clear the retransmit timer and round-trip time measurement timer. These will be
started by tcp_output_really when the retransmit is actioned. *)
tt_rexmt := *;
t_rttseg := *;

(* Sequence number to retransmit—this is equal to the ack value in the duplicate ACK
segment *)
snd_nxt := ack;
(* Ensure the congestion window is large enough to allow one segment to be emitted *)
snd_cwnd := cb.t_maxseg)) andThen

(* Attempt to create a segment for output using the modified control block (this is all a relational monad
idiom) *)
mlift_tcp_output_perhaps_or_fail ticks arch rttab ifds andThen

(* Finally, update the congestion window to snd_ssthresh plus 3 maximum segment sizes (this is the artificial
inflation of RFC2581/2582 because it is known that the 3 segments that generated the 3 duplicate acknowl-
edgments are received and no longer consuming network resource. Also put snd_nxt back to its previous
value. *)
modify_cb(λcb'.cb' ( [ snd_cwnd := cb'.snd_ssthresh + cb.t_maxseg * t_dupacks';
snd_nxt := max cb.snd_nxt cb'.snd_nxt ])) andThen
stop (* no need to process the segment any further *)

else assert_failure“di3_ackstuff” (* Believed to be impossible—here for completion and safety *)

else if ack ≤ cb.snd_una ∧ ¬maybe_dup_ack then
(* Have received an old (would use the word ”duplicate” if it did not have a special meaning) ACK and it is
neither a duplicate ACK nor the ACK of a new sequence number thus just clear the duplicate ACK counter. *)
modify_cb(λcb'.cb' ( [ t_dupacks := 0 ]))

else (* Must be: ack > cb.snd_una *)
(* This is the ACK of a new sequence number—this case is handled by the auxiliary function
di3_newackstuff (p295) *)
di3_newackstuff tcp_sock_0 seg ourfinisacked arch rttab ifds ticks
)

```

– *deliver_in_3* data processing :

```
di3_datastuff_really the_ststuff tcp_sock_0 seg bsd_fast_path arch =
```

```
(* Pull some fields out of the segment *)
```

```
let ACK = seg.ACK in
```

```
let FIN = seg.FIN in
```

```
let PSH = seg.PSH in
```

```
let URG = seg.URG in
```

```
let ack = tcp_seq_flip_sense seg.ack in
```

```
let urp = w2n seg.urp in
```

```
let data = seg.data in
```

```
let seq = tcp_seq_flip_sense seg.seq + (if seg.SYN then 1 else 0) in
```

```
(* Pull out the senders advertised window and apply the sender’s scale factor *)
```

```
let win = w2n seg.win << (tcp_sock_0).cb.snd_scale in
```

```
(* Get the socket’s control block using the monadic state accessor get_cb. Process the segments data and possibly
update the send window *)
```

```
(get_sock λsock.
```

```
let tcp_sock = tcp_sock_of sock in
```

```
let cb = tcp_sock.cb in
```

(* Trim segment to be within the receive window *)

(* Trim duplicate data from the left edge of *data*, i.e., data before *cb.rcv_nxt*. Adjust *seq*, *URG* and *urp* in respect of left edge trimming. If the urgent data has been trimmed from the segment's data, *URG* is cleared also. Note: the urgent pointer always points to the byte immediately following the urgent byte and is relative to the start of the segment's data. An urgent pointer of zero signifies that there is no urgent data in the segment. *)

```
let trim_amt_left = if cb.rcv_nxt > seq then min(num(cb.rcv_nxt - seq))(length data)
                    else 0 in
let data_trimmed_left = DROP trim_amt_left data in
let seq_trimmed = seq + trim_amt_left in (* Trimmed data starts at seq_trimmed *)
let urp_trimmed = if urp > trim_amt_left then urp - trim_amt_left else 0 in
let URG_trimmed = if urp_trimmed ≠ 0 then URG else F in
```

(* Trim any data outside the receive window from the right hand edge. If all the data is within the window and the *FIN* flag is set then the *FIN* flag is valid and should be processed. Note: this trimming may remove urgent data from the segment. The urgent pointer and flag are not cleared here because there is still urgent data to be received, but now in a future segment. *)

```
let data_trimmed_left_right = TAKE cb.rcv_wnd data_trimmed_left in
let FIN_trimmed = if data_trimmed_left_right = data_trimmed_left then FIN else F in
```

(* Processing of urgent (OOB) data: *)

(* We have a valid urgent pointer iff the trimmed segment has its urgent flag set with a non-zero urgent pointer, and the urgent pointer plus the length of the receive queue is less than or equal to *SB_MAX*. The last condition is imposed by FreeBSD, supposedly to prevent *soreceive* from crashing (although we cannot identify why it might crash). *)

```
let urp_valid = (URG_trimmed ∧ urp_trimmed > 0 ∧ urp_trimmed + length tcp_sock.rcvq ≤ SB_MAX) in
```

(* This is a new urgent pointer, i.e., it is greater than any previous one stored in *cb.rcv_up*. Note: the urgent pointer is relative to the sequence number of a segment *)

```
let urp_advanced = (urp_valid ∧ (seq_trimmed + urp_trimmed > cb.rcv_up)) in
```

(* The urgent pointer lies within segment *seq* and the socket is not set to do inline delivery, therefore it is possible to pull out the urgent byte from the stream *)

```
let can_pull = (urp_valid ∧
               urp_trimmed ≤ length data_trimmed_left_right ∧ sock.sf.b(SO_OOBINLINE) = F) in
```

(* Build trimmed segment to place on reassembly queue. If urgent data is in this segment and the socket is not doing inline delivery (and hence the urgent byte is stored in *iobc*), remove the urgent byte from the segment's data so that it does not get placed in the receive queue, and set *spliced_urp* to the sequence number of the urgent byte. *)

```
let rseq = ( seq := seq_trimmed;
            spliced_urp := if can_pull then ↑(cb.rcv_nxt + urp_trimmed - 1) else *;
            FIN := FIN_trimmed;
            data := if can_pull then
                    (TAKE(urp - 1) data_trimmed_left_right) @ (DROP urp data_trimmed_left_right)
                  else data_trimmed_left_right
            ) in
```

(* Perform a monadic socket state update *)

```
modify_tcp_sock(λs.s
  ( cb := s.cb
  ( (* If the segment's urgent pointer is valid and advances the urgent pointer, update rcv_up with
    the new absolute pointer, otherwise just pull it along with the left hand edge of the receive
    window. Note: an earlier segment may have set rcv_up to point somewhere into a future
    segment. The use of max ensures that the pointer is not accidentally overwritten until the
    future segment arrives. *)
    (* FreeBSD does not pull rcv_up along in the fast path; this is a bug *)
    rcv_up := (if urp_advanced then seq_trimmed + urp_trimmed
              else max cb.rcv_up cb.rcv_nxt)
    onlywhen ¬(bsd_arch arch ∧ bsd_fast_path));
```

(* If the urgent pointer is valid and advances the urgent pointer, update *rcvurp*—the socket's receive queue urgent data index—to be the index into the receive queue where the new urgent data will be stored. Note: the subtraction of 1 is correct because *rcvurp* points to the location where the urgent byte is stored not the byte immediately following the urgent byte (as is the convention for the *urp* field in the TCP header). *)

```
rcvurp := (↑(length tcp_sock.rcvq +
             num(seq_trimmed + urp_trimmed - cb.rcv_nxt - 1)))
```

onlywhen *urp_advanced*;

(* If the segment's urgent pointer is valid, the urgent data is within this segment and the socket is not doing inline delivery of urgent data, pull out the urgent byte into *iobc*. If the urgent data is within a future segment set *iobc* to NO_OOBDATA to signify that the urgent data is not available yet, otherwise leave *iobc* alone if the urgent pointer is not valid. *)

```
iobc := (if can_pull then OOBDATA(EL(urp - 1)
                                   data_trimmed_left_right)
        else NO_OOBDATA)
```

onlywhen *urp_valid*

}) andThen

(* Processing of non-urgent data. There are 6 cases to consider: *)

(chooseM{**F**; **T**} λ *FIN_reass*.

(* Case (1) The segment contains new in-order, in-window data possibly with a *FIN* and the receive window is not closed. Note: it is possible that the segment contains just one byte of OOB data that may have already been pulled out into *iobc* if OOB delivery is out-of-line. In which case, the below must still be performed even though no data is contributed to the reassembly buffer in order that *rcv_nxt* is updated correctly (because a byte of urgent data consumes a byte of sequence number space). This is why *data_trimmed_left_right* is used rather than *data_deobed* in some of the conditions below. *)

```
(if seq_trimmed = cb.rcv_nxt ∧
   seq_trimmed + length data_trimmed_left_right + (if FIN_trimmed then 1 else 0) > cb.rcv_nxt ∧
   cb.rcv_wnd > 0 then
```

(* Only need to acknowledge the segment if there is new in-window data (including urgent data) or a valid *FIN* *)

```
let have_stuff_to_ack = (data_trimmed_left_right ≠ [] ∨ FIN_trimmed) in
```

(* If the socket is connected, has data to *ACK* but no *FIN* to *ACK*, the reassembly queue is empty, the socket is not currently within a bad retransmit window and an *ACK* is not already being delayed, then delay the *ACK*. *)

```
let delay_ack = (tcp_sock.st ∈ {ESTABLISHED; CLOSE_WAIT; FIN_WAIT_1;
                                CLOSING; LAST_ACK; FIN_WAIT_2} ∧
```

```
   have_stuff_to_ack ∧
   ¬FIN_trimmed ∧
   cb.t_seqq = [] ∧
   ¬cb.tf_rxwin0sent ∧
   cb.tt_delack = *) in
```

(* Check to see whether any data or a *FIN* can be reassembled. *tcp_reass* returns the set of all possible reassemblies, one of which is chosen non-deterministically here. Note: a *FIN* can only be reassembled once all the data has been reassembled. The *len* result from *tcp_reass* is the length of the reassembled data, *data_reass*, plus the length of any out-of-line urgent data that is not included in the reassembled data but logically occurs within it. This is to ensure that control block variables such as *rcv_nxt* are incremented by the correct amount, i.e., by the amount of data (whether urgent or not) received successfully by the socket. See *tcp_reass* (p100) for further details. *)

```
let rseqq = rseq :: cb.t_seqq in
```

```
(chooseM(tcp_reass cb.rcv_nxt rseqq) λ (data_reass, len, FIN_reass0).
```

(* Length (in sequence space) of reassembled data, counting a *FIN* as one byte and including any out-of-line urgent data previously removed *)

```
let len_reass = len + (if FIN_reass0 then 1 else 0) in
```

(* Add the reassembled data to the receive queue and increment *rcv_nxt* to mark the sequence number of the byte past the last byte in the receive queue*)

```
let rcvq' = tcp_sock.rcvq @ data_reass in
```

let $rcv_nxt' = cb.rcv_nxt + len_reass$ **in** (* includes oob bytes as they occupy sequence space *)

(* Prune the receive queue of any data or *FIN*s that were reassembled, keeping all segments that contain data at or past sequence number $cb.rcv_nxt + len_reass$. *)

let $t_segq' = tcp_reass_prune\ rcv_nxt'\ rsegq$ **in**

(* Reduce the receive window in light of the data added to the receive queue. Do not include out-of-line urgent data because it does not store data in the receive queue. *)

let $rcv_wnd' = cb.rcv_wnd - \mathbf{length}\ data_reass$ **in**

(* Hack: assertion used to share values with later conditions *)

assert($FIN_reass = FIN_reass0$) **andThen**

(* Update the socket state *)

$modify_tcp_sock(\lambda s.s$

$\llbracket rcvq := rcvq';$ (* the updated receive queue *)

$cb := s.cb$

\llbracket (* Start the delayed ack timer if decided to earlier, i.e., $delay_ack = \mathbf{T}$. *)

$tt_delack := \uparrow(((\))_{fast_timer}\ TCPTV_DELACK)$ **onlywhen** $delay_ack$;

 (* Set if not delaying an *ACK* and have stuff to *ACK* *)

$tf_shouldacknow := \neg delay_ack$ **onlywhen** $have_stuff_to_ack$;

$t_segq := t_segq'$; (* updated reassembly queue, post-pruning *)

$rcv_nxt := rcv_nxt'$;

$rcv_wnd := rcv_wnd'$

\rrbracket

\rrbracket

)(* chooseM *)

(* Case (2) The segment contains new out-of-order in-window data, possibly with a *FIN*, and the receive window is not closed. Note: it may also contain in-window urgent data that may have been pulled out-of-line but still require processing to keep reassembly happy. *)

else if $seq_trimmed > cb.rcv_nxt \wedge seq_trimmed < cb.rcv_nxt + cb.rcv_wnd \wedge$

$\mathbf{length}\ data_trimmed_left_right + (\mathbf{if}\ FIN_trimmed\ \mathbf{then}\ 1\ \mathbf{else}\ 0) > 0 \wedge$

$cb.rcv_wnd > 0$ **then**

(* Hack: assertion used to share values with later conditions *)

assert($FIN_reass = \mathbf{F}$) **andThen**

(* Update the socket's TCP control block state *)

$modify_cb(\lambda cb.cb$ \llbracket (* Add the segment to the reassembly queue *)

$t_segq := rseg :: cb.t_segq$;

 (* Acknowledge out-of-order data immediately (per RFC2581 Ch4.2) *)

$tf_shouldacknow := \mathbf{T}$

\rrbracket

(* Case (3) The segment is a pure *ACK* segment (contains no data) (and must be in-order). *)

(* Invariant here that $seq_trimmed = seq$ if segment is a pure *ACK*. Note: the length of the original segment (not the trimmed segment) is used in the guard to ensure this really was a pure *ACK* segment. *)

else if $ACK \wedge seq_trimmed = cb.rcv_nxt \wedge \mathbf{length}\ data + (\mathbf{if}\ FIN\ \mathbf{then}\ 1\ \mathbf{else}\ 0) = 0$ **then**

(* Hack: assertion used to share values with later conditions *)

assert($FIN_reass = \mathbf{F}$) (* Have not received a *FIN* *)

(* Case (4) Segment contained no useful data—was a completely old segment. Note: the original fields from the segment, i.e., seq , $data$ and FIN are used in the guard below—the trimmed variants are useless here! *)

(* Case (5) Segment is a window probe. Note: the original fields from the segment, i.e., $data$ and FIN are used in the guard below—the trimmed variants are useless here! *)

(* Case (6) Segment is completely beyond the window and is not a window probe *)

else if $(seq < cb.rcv_nxt \wedge seq + \mathbf{length}\ data + (\mathbf{if}\ FIN\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \leq cb.rcv_nxt) \vee$ (* (4) *)

$(seq_trimmed = cb.rcv_nxt \wedge cb.rcv_wnd = 0 \wedge$


```

    length data + (if FIN then 1 else 0) > 0) ∨ (* (5) *)
  T then (* (6) *)

    (* Hack: assertion used to share values with later conditions *)
    assert(FIN_reass = F) andThen (* Definitely false—segment is outside window *)

    (* Update socket's control block to assert that an ACK segment should be sent now. *)
    (* Source: TCPIPv2p959 says "segment is discarded and an ack is sent as a reply" *)
    modify_cb(λcb.cb ⟨ tf_shouldacknow := T ⟩)

else
  assert_failure "di3_datastuff" (* impossible *)

) andThen

(* Finished processing the segment's data *)
(* Thread the reassembled FIN flag through to di3_ststuff *)
the_ststuff FIN_reass

)(* chooseM FIN_reass *)
)(* get_sock \sock *)

|-----|

|-----|

- deliver_in_3 data processing :
di3_datastuff the_ststuff tcp_sock_0 seg ourfinisacked arch =
  (* Pull some fields out of the segment *)
  let ACK = seg.ACK in
  let FIN = seg.FIN in
  let PSH = seg.PSH in
  let URG = seg.URG in
  let ack = tcp_seq_flip_sense seg.ack in
  let urp = w2n seg.urp in
  let data = seg.data in
  let seq = tcp_seq_flip_sense seg.seq + (if seg.SYN then 1 else 0) in
  let win = w2n seg.win << (tcp_sock_0).cb.snd_scale in

  get_sock λsock.
  let tcp_sock = tcp_sock_of sock in
  let cb = tcp_sock.cb in

  (* Various things do not happen if BSD processes the segment using its header prediction (fast-path) code. Header
  prediction occurs only in the ESTABLISHED state, with segments that have only ACK and/or PSH flags set, are
  in-order, do not contain a window update, when data is not being retransmitted (no congestion is occurring) and either:
  (a) the segment is a valid pure ACK segment of new data, less than three duplicate ACKs have been received and the
  congestion window is at least as large as the send window, or (b) the segment contains new data, does not acknowledge
  any new data, the segment reassembly queue is empty and there is space for the segment's data in the socket's receive
  buffer. *)
  let bsd_fast_path = ((tcp_sock.st = ESTABLISHED) ∧ ¬seg.SYN ∧ ¬FIN ∧ ¬seg.RST ∧
    ¬URG ∧ ACK ∧ seq = cb.rcv_nxt ∧ cb.snd_wnd = win ∧
    cb.snd_max = cb.snd_nxt ∧ (
      ack > cb.snd_una ∧ ack ≤ cb.snd_max ∧
      cb.snd_cwnd ≥ cb.snd_wnd ∧ cb.t_dupacks < 3)
    ∨
    (ack = cb.snd_una ∧ cb.t_seqq = [] ∧
      (length data) <
      (sock.sf.n(SO_RCVBUF) - length tcp_sock.rcvq)))) in

```

(* Update the send window using the received segment if the segment will not be processed by BSD's fast path, has the *ACK* flag set, is not to the right of the window, and either:

(a) the last window update was from a segment with sequence number less than *seq*, i.e., an older segment than the current segment, or

(b) the last window update was from a segment with sequence number equal to *seq* but with an acknowledgement number less than *ack*, i.e., this segment acknowledges newer data than the segment the last window update was taken from, or

(c) the last window update was from a segment with sequence number equal to *seq* and acknowledgement number equal to *ack*, i.e., a segment similar to that the previous update came from, but this segment contains a larger window advertisement than was previously advertised, or

(d) this segment is the third segment during connection establishment (state is *SYN_RECEIVED*) and does not have the *FIN* flag set. *)

```
let update_send_window = (¬bsd_fast_path ∧ seq.ACK ∧ seq ≤ cb.rcv_nxt + cb.rcv_wnd ∧
  (cb.snd_wl1 < seq ∨
   (cb.snd_wl1 = seq ∧
    (cb.snd_wl2 < ack ∨ cb.snd_wl2 = ack ∧ win > cb.snd_wnd))) ∨
  (tcp_sock.st = SYN_RECEIVED ∧ ¬FIN)) in (* This replaces BSD's snd_wl1
  := seq-1 hack; should perhaps
  be ¬FIN_reass *)
```

```
let seq_trimmed = max seq(min cb.rcv_nxt(seq + length data)) in
```

(* Write back the window updates *)

```
modify_cb(λcb.cb { snd_wnd := win onlywhen update_send_window;
  snd_wl1 := seq_trimmed onlywhen update_send_window;
  snd_wl2 := ack onlywhen update_send_window
  (* persist timer will be set by deliver_out_1 if this updates the window to zero and there is data
  to send *)
}) andThen
```

(* If in *TIME_WAIT* or will transition to it from *CLOSING*, ignore any *URG*, data, or *FIN*. Note that in *FIN_WAIT_1* or *FIN_WAIT_2*, we still process data, even if *ourfinisacked*. *)

```
if tcp_sock.st = TIME_WAIT ∨ (tcp_sock.st = CLOSING ∧ ourfinisacked) then
```

(* pull along urgent pointer *)

```
modify_cb(λcb.cb { rcv_up := max cb.rcv_up cb.rcv_nxt}) andThen
the_ststuff F
```

else

```
di3_datastuff_really the_ststuff tcp_sock_0 seg bsd_fast_path arch
```

– *deliver_in_3* **TCP state change processing** :

```
di3_ststuff FIN_reass ourfinisacked ack =
```

(* The entirety of this function is an encoding of the TCP State Transition Diagram (as it is, not as it is traditionally depicted) post-*SYN_SENT* state. It specifies for given start state and set of conditions (all or some of which are affected by the processing of the current segment), which state the TCP socket should be moved into next *)

(* Get the TCP socket using the monadic state accessor *get_cb*. *)

```
(get_sock λsock.
```

```
let cb = (tcp_sock_of sock).cb in (* ...and its control block *)
```

(* Several of the encoded transitions (below) require the socket to be moved into the *TIME_WAIT* state, in which case the *2MSL* timer is started, all other timers are cancelled and the socket's state is changed to *TIME_WAIT*. This common idiom is defined monadically as a function here *)

```
let enter_TIME_WAIT =
```

```
modify_tcp_sock(λs.s
  { st := TIME_WAIT;
    cb := s.cb
```

```

    ⟨ tt_2msl := ↑((( ))_slow_timer(2*TCPTV_MSL));
      tt_rexmt := *;
      tt_keep := *;
      tt_delack := *;
      tt_conn_est := *;
      tt_fin_wait_2 := *
    ⟩
  ⟩) in

```

(* If the processing of the current segment has led to *FIN_reass* being asserted then the whole data stream from the other end has been received and reconstructed, including the final *FIN* flag. The socket should have its read-half flagged as shut down, i.e., *cantrcvmore* = **T**, otherwise the socket is not modified. *)

```

(if FIN_reass then
  modify_sock(λs.s ⟨ cantrcvmore := T ⟩)
else cont) andThen

```

(* State Transition Diagram encoding: *)

(* The state transition encoding, case-split on the current state and whether a *FIN* from the remote end has been reassembled *)

```

case ((tcp_sock_of sock).st, FIN_reass) of

```

```

(SYN_RECEIVED, F) → (* In SYN_RECEIVED and have not received a FIN *)

```

```

  if ack ≥ cb.iss + 1 then

```

```

    (* This socket's initial SYN has been acknowledged *)

```

```

    modify_tcp_sock(λs.s

```

```

      ⟨ st := if ¬sock.cantsndmore then

```

```

        ESTABLISHED (* socket is now fully connected *)

```

```

      else

```

```

        (* The connecting socket had its write-half shutdown by shutdown() forcing a FIN to be emitted to
        the other end *)

```

```

        if ourfinisacked then

```

```

          (* The emitted FIN has been acknowledged *)

```

```

          FIN_WAIT_2

```

```

        else

```

```

          (* Still waiting for the emitted FIN to be acknowledged *)

```

```

          FIN_WAIT_1

```

```

      ⟩)

```

```

  else

```

```

    (* Not a valid path *)

```

```

    stop ||

```

```

(SYN_RECEIVED, T) → (* In SYN_RECEIVED and have received a FIN *)

```

```

  (* Enter the CLOSE_WAIT state, missing out ESTABLISHED *)

```

```

  modify_tcp_sock(λs.s ⟨ st := CLOSE_WAIT ⟩) ||

```

```

(ESTABLISHED, F) → (* In ESTABLISHED and have not received a FIN *)

```

```

  (* Doing common-case data delivery and acknowledgements. Remain in ESTABLISHED. *)

```

```

  cont ||

```

```

(ESTABLISHED, T) → (* In ESTABLISHED and received a FIN *)

```

```

  (* Move into the CLOSE_WAIT state *)

```

```

  modify_tcp_sock(λs.s ⟨ st := CLOSE_WAIT ⟩) ||

```

```

(CLOSE_WAIT, F) → (* In CLOSE_WAIT and have not received a FIN *)

```

```

  (* Do nothing and remain in CLOSE_WAIT. The socket has its receive-side shut down due to the FIN it received
  previously from the remote end. It can continue to emit segments containing data and receive acknowledgements
  back until such a time that it closes down and emits a FIN *)

```

```

cont ||

(CLOSE_WAIT, T) → (* In CLOSE_WAIT and received (another) FIN *)
(* The duplicate FIN will have had a new sequence number to be valid and reach this point; RFC793 says "ignore"
it so do not change state! If it were a duplicate with the same sequence number as the previously accepted FIN,
then the deliver_in_3 acknowledgement processing function di3_ackstuff would have dropped it. *)
cont ||

(FIN_WAIT_1, F) → (* In FIN_WAIT_1 and have not received a FIN *)
(* This socket will have emitted a FIN to enter FIN_WAIT_1. *)
if ourfinisacked then
  (* If this socket's FIN has been acknowledged, enter state FIN_WAIT_2 and start the FIN_WAIT_2 timer.
  The timer ensures that if the other end has gone away without emitting a FIN and does not transmit any more
  data the socket is closed rather left dangling. *)
  modify_tcp_sock(λs.s
    ⟨ st := FIN_WAIT_2;
      cb := s.cb
      ⟨ tt_fin_wait_2 := ↑((( )_slow_timer TCPTV_MAXIDLE)
        onlywhen sock.cantrcvmore (* believe always true *)
      ⟩
    ⟩
  )
else
  (* If this socket's FIN has not been acknowledged then remain in FIN_WAIT_1 *)
  cont ||

(FIN_WAIT_1, T) → (* In FIN_WAIT_1 and received a FIN *)
if ourfinisacked then
  (* ...and this socket's FIN has been acknowledged then the connection has been closed successfully so enter
  TIME_WAIT. Note: this differs slightly from the behaviour of BSD which momentarily enters the
  FIN_WAIT_2 and after a little more processing enters TIME_WAIT *)
  enter_TIME_WAIT
else
  (* If this socket's FIN has not been acknowledged then the other end is attempting to close the connection
  simultaneously (a simultaneous close). Move to the CLOSING state *)
  modify_tcp_sock(λs.s ⟨ st := CLOSING ⟩) ||

(FIN_WAIT_2, F) → (* In FIN_WAIT_2 and have not received a FIN *)
(* This socket has previously emitted a FIN which has already been acknowledged. It can continue to receive
data from the other end which it must acknowledge. During this time the socket should remain in FIN_WAIT_2
until such a time that it receives a valid FIN from the remote end, or if no activity occurs on the connection the
FIN_WAIT_2 timer will fire, eventually closing the socket *)
cont ||

(FIN_WAIT_2, T) → (* In FIN_WAIT_2 and have received a FIN *)
(* Connection has been shutdown so enter TIME_WAIT *)
enter_TIME_WAIT ||

(CLOSING, F) → (* In CLOSING and have not received a FIN *)
if ourfinisacked then
  (* If this socket's FIN has been acknowledged (common-case), enter TIME_WAIT as the connection has been
  successfully closed *)
  enter_TIME_WAIT
else
  (* Otherwise, the other end has not yet received or processed the FIN emitted by this socket. Remain in
  the CLOSING state until it does so. Note: if the previously emitted FIN is not acknowledged this socket's
  retransmit timer will eventually fire causing retransmission of the FIN. *)
  cont ||

```

```

(CLOSING, T) → (* In CLOSING and have received a FIN *)
  (* The received FIN is a duplicate FIN with a new sequence number so as per RFC793 is ignored – if it were a
  duplicate with the same sequence number as the previously accepted FIN, then the deliver_in_3 acknowledgement
  processing function di3_ackstuff would have dropped it. *)
  if ourfinisacked then
    (* If this socket's FIN has been acknowledged then the connection is now successfully closed, so enter
    TIME_WAIT state *)
    enter_TIME_WAIT
  else
    (* Otherwise, ignore the new FIN and remain in the same state *)
    cont ||

(LAST_ACK, F) → (* In LAST_ACK and have not received a FIN *)
  (* Remain in LAST_ACK until this socket's FIN is acknowledged. Note: eventually the retransmit timer will
  fire forcing the FIN to be retransmitted. *)
  cont ||

(LAST_ACK, T) → (* In LAST_ACK and have received a FIN *)
  (* This transition is handled specially at the end of di3_newackstuff at which point processing stops, thus this
  transition is not possible *)
  assert_failure“di3_ststuff” (* impossible *) ||

(TIME_WAIT, F) → (* In TIME_WAIT and have not received a FIN *)
  (* Remaining in TIME_WAIT until the 2MSL timer expires *)
  cont ||

(TIME_WAIT, T) → (* In TIME_WAIT and have received a FIN *)
  (* Remaining in TIME_WAIT until the 2MSL timer expires *)
  cont
)

```

– *deliver_in_3* socket update processing :

```
di3_socks_update sid socks socks' =
```

```
let sock_1 = socks[sid] in
∃tcp_sock_1.
TCP_PROTO(tcp_sock_1) = sock_1.pr ∧
```

(* Socket *sock_1* referenced by identifier *sid* has just finished connection establishment and either there is another socket with *sock_1* on its pending connections queue and this is the completion of a passive open, or there is not another socket and this is the completion of a simultaneous open. See the inline comment in *deliver_in_3* (p292) for further details. *)

```
let interesting = λsid'.
  sid' ≠ sid ∧
  case (socks[sid']).pr of
    UDP_PROTO udp_sock → F
  || TCP_PROTO(tcp_sock') →
    case tcp_sock'.lis of
      * → F
    || ↑ lis →
      sid ∈ lis.q0 in
```

```
let interesting_sids = (dom(socks)) ∩ interesting in
```

```
if interesting_sids ≠ {} then
```

(* There exists another socket *sock'* that is listening and has socket *sock_1* referenced by *sid* on its queue of incomplete connections *lis.q₀*. *)

$\exists sid' \text{ sock}' \text{ tcp_sock}' \text{ lis } q0L \text{ } q0R.$

$sid' \in \text{interesting_sids} \wedge$

$\text{sock}' = \text{socks}[sid'] \wedge$

$\text{sock}'.\text{pr} = \text{TCP_PROTO } \text{tcp_sock}' \wedge$

$sid' \neq \text{sid} \wedge$

$\text{tcp_sock}'.\text{lis} = \uparrow \text{lis} \wedge$

$\text{lis}.q_0 = q0L @ (\text{sid} :: q0R) \wedge$

(* Choose non-deterministically whether there is room on the queue of completed connections *)

choose *ok* :: accept_incoming_q *lis*.

if *ok* **then**

(* If there is room, then remove socket *sid* from the queue of incomplete connections and add it to the queue of completed connections. *)

let $lis' = lis \{ q_0 := q0L @ q0R;$
 $q := \text{sid} :: lis.q \}$ **in**

(* Update the newly connected sockets receive window *)

let *rcv_window* = calculate_bsd_rcv_wnd *sock_1.sf tcp_sock_1* **in**

(* BSD bug - *rcv_adv* gets incorrectly set using the old value of *rcv_wnd*, as this is done by the syncache, which is called from *tcp_input()* before the *rcv_wnd* update takes place. Note that we have the following: SYN_SENT->ESTABLISHED => update *rcv_wnd* then *rcv_adv* SYN_RCVD->ESTABLISHED => update *rcv_adv* then *rcv_wnd* *)

let $cb' = \text{tcp_sock_1}.cb \{ \text{rcv_wnd} := \text{rcv_window};$
 $\text{rcv_adv} := \text{tcp_sock_1}.cb.\text{rcv_next} + \text{tcp_sock_1}.cb.\text{rcv_wnd} \}$ **in**

(* Update both the newly connected socket and the listening socket *)

$\text{socks}' = \text{socks} \oplus$
 $[(\text{sid}, \text{sock_1} \{ \text{pr} := \text{TCP_PROTO}(\text{tcp_sock_1} \{ \text{cb} := \text{cb}' \}) \})];$
 $(\text{sid}', \text{sock}' \{ \text{pr} := \text{TCP_PROTO}(\text{tcp_sock}' \{ \text{lis} := \uparrow \text{lis}' \}) \})]$

else

(* ...otherwise there is no room on the listening socket's completed connections queue, so drop the newly connected socket and remove it from the listening socket's queue of incomplete connections. Note: the dropped connection is not sent a *RST* but a *RST* is sent upon receipt of further segments from the other end as the socket entry has gone away. *)

(* Note that the above note needs to be verified by testing. *)

let $lis' = lis \{ q_0 := q0L @ q0R \}$ **in**
 $\text{socks}' = \text{socks} \oplus (\text{sid}', \text{sock}' \{ \text{pr} := \text{TCP_PROTO}(\text{tcp_sock}' \{ \text{lis} := \uparrow \text{lis}' \}) \})$

else

(* There is no such socket with socket *sid* on its queue of incomplete connections, thus socket *sid* was involved in a simultaneous open. Do not update any socket. *)

$\text{socks}' = \text{socks}$

deliver_in_3a **tcp: network nonurgent** Receive data with invalid checksum or offset

$h \{ \text{socks} := \text{socks}; \quad \xrightarrow{T} \quad h \{ \text{socks} := \text{socks};$
 $iq := iq \} \quad \quad \quad iq := iq' \}$

(* **Summary:** This rule is a placeholder for the case where a received segment has an invalid checksum or offset, in which case implementations should drop it on the floor. The model of TCP segments does not contain checksum or offset, however, hence the **F** below. *)

$sid \in \text{dom}(\text{socks}) \wedge$

$\text{sock_0} = \text{socks}[sid] \wedge$

$\text{sock_0}.is_1 = \uparrow i_1 \wedge \text{sock_0}.ps_1 = \uparrow p_1 \wedge \text{sock_0}.is_2 = \uparrow i_2 \wedge \text{sock_0}.ps_2 = \uparrow p_2 \wedge$

$sock_0.pr = TCP_PROTO(tcp_sock_0) \wedge$

$dequeue_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$

$(\exists win_urp_ws_discard\ mss_discard.$

$win = \mathbf{w2n}\ win_ \ll tcp_sock_0.cb.snd_scale \wedge$

$urp = \mathbf{w2n}\ urp_ \wedge$

$seg = \langle$

$is_1 := \uparrow i_2;$

$is_2 := \uparrow i_1;$

$ps_1 := \uparrow p_2;$

$ps_2 := \uparrow p_1;$

$seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);$

$ack := tcp_seq_flip_sense(ack : tcp_seq_local);$

$URG := URG;$

$ACK := ACK;$

$PSH := PSH;$

$RST := \mathbf{F};$

$SYN := \mathbf{F};$

$FIN := FIN;$

$win := win_;$

$ws := ws_discard;$

$urp := urp_;$

$mss := mss_discard;$

$ts := ts;$

$data := data$

$\rangle) \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly *)

$tcp_sock_0.st \notin \{CLOSED; LISTEN; SYN_SENT\} \wedge$

$tcp_sock_0.st \in \{SYN_RECEIVED; ESTABLISHED; CLOSE_WAIT; FIN_WAIT_1; FIN_WAIT_2;$
 $CLOSING; LAST_ACK; TIME_WAIT\} \wedge$

F (* invalid checksum or offset *)

deliver_in_3b **tcp: network nonurgent** Receive data after process has gone away

$h \langle socks := socks; \quad \xrightarrow{T} \quad h \langle socks := socks';$
 $iq := iq; \quad \quad \quad iq := iq';$
 $oq := oq; \quad \quad \quad oq := oq';$
 $bndlm := bndlm \rangle \quad \quad \quad bndlm := bndlm' \rangle$

(* **Summary:** if data arrives after the process associated with a socket has gone away, close socket and emit RST segment. *)

$sid \in \mathbf{dom}(socks) \wedge$

$sock_0 = socks[sid] \wedge$

$sock_0.is_1 = \uparrow i_1 \wedge sock_0.ps_1 = \uparrow p_1 \wedge sock_0.is_2 = \uparrow i_2 \wedge sock_0.ps_2 = \uparrow p_2 \wedge$

$sock_0.pr = TCP_PROTO(tcp_sock_0) \wedge$

$dequeue_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$

$(\exists win_urp_ws_discard\ mss_discard.$

$win = \mathbf{w2n}\ win_ \ll tcp_sock_0.cb.snd_scale \wedge$

$urp = \mathbf{w2n}\ urp_ \wedge$

```

seg =⟦
  is1 := ↑ i2;
  is2 := ↑ i1;
  ps1 := ↑ p2;
  ps2 := ↑ p1;
  seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);
  ack := tcp_seq_flip_sense(ack : tcp_seq_local);
  URG := URG;
  ACK := ACK;
  PSH := PSH;
  RST := F;
  SYN := F;
  FIN := FIN;
  win := win_;
  ws := ws_discard;
  wrp := wrp_;
  mss := mss_discard;
  ts := ts;
  data := data
  ⟧) ∧

```

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

(* test that this is data arriving after process has gone away *)

```

tcp_sock_0.st ∈ {FIN_WAIT_1; CLOSING; LAST_ACK; FIN_WAIT_2; TIME_WAIT} ∧
sock_0.fid = * ∧
seq + length data > tcp_sock_0.cb.rcv_nxt ∧

```

(* close socket and emit RST segment *)

```

socks' = socks ⊕ (sid, tcp_close h.arch sock_0) ∧
dropwithreset_ignore_fail seq h.arch h.ifds h.rttab(ticks_of h.ticks)
  BANDLIM_UNLIMITED bndlm bndlm' outsegs ∧
enqueue_oq_list_qinfo(oq, outsegs, oq')

```

deliver_in_3c **tcp: network nonurgent** **Receive stupid ACK or LAND DoS in SYN_RECEIVED state**

```

h ⟦socks := socks;     $\xrightarrow{T}$     h ⟦socks := socks';
iq := iq;                iq := iq';
oq := oq;                oq := oq';
bndlm := bndlm ⟧        bndlm := bndlm' ⟧

```

(* **Summary:** if we receive a stupid ACK or a LAND DoS in SYN_RECEIVED state then update timers and emit a RST appropriately. *)

```

sid ∈ dom(socks) ∧
sock_0 = socks[sid] ∧
sock_0.is1 = ↑ i1 ∧ sock_0.ps1 = ↑ p1 ∧ sock_0.is2 = ↑ i2 ∧ sock_0.ps2 = ↑ p2 ∧
sock_0.pr = TCP_PROTO(tcp_sock_0) ∧

```

```

dequeue_iq(iq, iq', ↑(TCP seg)) ∧

```

```

(∃ win_ wrp_ ws_discard mss_discard.
win = w2n win_ << tcp_sock_0.cb.snd_scale ∧
wrp = w2n wrp_ ∧
seg =⟦

```



```

    is1 := ↑ i2;
    is2 := ↑ i1;
    ps1 := ↑ p2;
    ps2 := ↑ p1;
    seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);
    ack := tcp_seq_flip_sense(ack : tcp_seq_local);
    URG := URG;
    ACK := ACK;
    PSH := PSH;
    RST := F;
    SYN := F;
    FIN := FIN;
    win := win_;
    ws := ws_discard;
    wrp := wrp_;
    mss := mss_discard;
    ts := ts;
    data := data
  ]) ∧

```

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

(* test for stupid ACK in SYN_RECEIVED, and for LAND DoS attack *)

```

tcp_sock_0.st = SYN_RECEIVED ∧
((ACK ∧ (ack ≤ tcp_sock_0.cb.snd_una ∨ ack > tcp_sock_0.cb.snd_max)) ∨
seq < tcp_sock_0.cb.irs) ∧

```

(* incoming segment; update timers *)

```

let (t_idletime', tt_keep', tt_fin_wait_2') = update_idle tcp_sock_0 in
let tcp_sock' = tcp_sock_0 { cb := tcp_sock_0.cb
    { t_idletime := t_idletime';
      tt_keep := tt_keep';
      tt_fin_wait_2 := tt_fin_wait_2' }} in
socks' = socks ⊕ (sid, sock_0 { pr := TCP_PROTO(tcp_sock') }) ∧

```

(* emit RST. See dropwithreset_ignore_fail (p120) and enqueue_oq_list_qinfo (p??). *)

```

dropwithreset_ignore_fail seq h.arch h.ifds h.rttab(ticks_of h.ticks)
  BANDLIM_UNLIMITED bndlm bndlm' outsegs ∧
enqueue_oq_list_qinfo(oq, outsegs, oq')

```

deliver_in_4 **tcp: network nonurgent** Receive and drop (silently) a non-sane or martian segment

$h \langle iq := iq \rangle \xrightarrow{\tau} h \langle iq := iq' \rangle$

(* **Summary:** Receive and drop any segment for this host that does not have sensible checksum or offset fields, or one that originates from a martian address. The first part of this condition is a placeholder, awaiting the day when we switch to a non-lossy segment representation, hence the **F**. *)

```

dequeue_iq(iq, iq', ↑(TCP seg)) ∧
seg.is2 = ↑ i2 ∧
is1 = seg.is1 ∧
i2 ∈ localLips(h.ifds) ∧
(F ∨ (* placeholder for segment checksum and offset field not sensible *))
¬(
  T ∧ (* placeholder for not a link-layer multicast or broadcast *)

```

$\neg(\text{is_broadmcast } h.\text{ifds } i_2) \wedge$ (* seems unlikely, since $i_1 \in \text{localLips } h.\text{ifds}$ *)
 $\neg(is_1 = *) \wedge$
 $\neg \text{is_broadmcast } h.\text{ifds}(\text{the } is_1)$
)
)

deliver_in_5 tcp: network nonurgent Receive and drop (maybe with RST) a sane segment that does not match any socket

$h \langle iq := iq; \quad \quad \quad \tau \rightarrow \quad h \langle iq := iq';$
 $oq := oq; \quad \quad \quad \quad \quad \quad \quad oq := oq';$
 $bndlm := bndlm \rangle \quad \quad \quad \quad \quad \quad \quad bndlm := bndlm' \rangle$

(* **Summary:** Receive and drop any segment for this host that does not match any sockets (but does have sensible checksum and offset fields). Typically, generate RST in response, computing *ack* and *seq* to supposedly make the other end see this as an 'acceptable ack'. *)

dequeue_iq(*iq*, *iq'*, ↑(TCP *seg*)) ∧

$seg.is_2 = \uparrow i_1 \wedge i_1 \in \text{localLips}(h.\text{ifds}) \wedge$
 $seg.ps_2 = \uparrow p_1 \wedge$
 $seg.is_1 \neq * \wedge seg.ps_1 \neq * \wedge$

T ∧ (* placeholder for segment checksum and offset field sensible *)

$\neg(\exists((\text{sid}, \text{sock}) :: h.\text{socks}) \text{tcp_sock}.$
 $\text{sock.pr} = \text{TCP_PROTO}(\text{tcp_sock}) \wedge$
 $\text{match_score}(\text{sock.is}_1, \text{sock.ps}_1, \text{sock.is}_2, \text{sock.ps}_2)$
 $\quad \quad \quad (\text{the } seg.is_1, seg.ps_1, \text{the } seg.is_2, seg.ps_2) > 0$
 $) \wedge$

dropwithreset *seg* *h.ifds*(ticks_of *h.ticks*) BANDLIM_RST_CLOSEDPORT *bndlm bndlm'* *outsegs'* ∧
 enqueue_and_ignore_fail *h.arch h.rttab h.ifds outsegs' oq oq'*

deliver_in_6 tcp: network nonurgent Receive and drop (silently) a sane segment that matches a CLOSED socket

$h \langle iq := iq \rangle \quad \tau \rightarrow \quad h \langle iq := iq' \rangle$

(* **Summary:** Receive and drop any segment for this host that does not match any sockets (but does have sensible checksum or offset fields).

Note that pathological segments where *is*₁, *ps*₁, or *ps*₂ are not set in the segment are not dealt with here but need to be. *)

dequeue_iq(*iq*, *iq'*, ↑(TCP *seg*)) ∧

$(\exists((\text{sid}, \text{sock}) :: h.\text{socks}) \text{tcp_sock}.$
 $\text{sock.pr} = \text{TCP_PROTO}(\text{tcp_sock}) \wedge$
 $\text{match_score}(\text{sock.is}_1, \text{sock.ps}_1, \text{sock.is}_2, \text{sock.ps}_2)$
 $\quad \quad \quad (\text{the } seg.is_1, seg.ps_1, \text{the } seg.is_2, seg.ps_2) > 0 \wedge$
 $\text{tcp_socket_best_match } h.\text{socks}(\text{sid}, \text{sock}) \text{seg } h.\text{arch} \wedge$
 $\text{tcp_sock.st} = \text{CLOSED}) \wedge$
 $seg.is_2 = \uparrow i_1 \wedge i_1 \in \text{localLips}(h.\text{ifds}) \wedge$
 \mathbf{T} (* placeholder for segment checksum and offset field sensible *)

deliver_in_7 **tcp: network nonurgent** Receive RST and zap non-{CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} socket

$$h \langle ts := ts \oplus (tid \mapsto (ts_{st})_d); \quad \xrightarrow{\tau} \quad h \langle ts := ts \oplus (tid \mapsto (ts_{st})_d);$$

$$socks := socks \oplus [(sid, sock)]; \quad socks := socks \oplus [(sid, sock)];$$

$$iq := iq \rangle \quad iq := iq' \rangle$$

(* **Summary:** receive RST and silently zap non-{CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} socket *)

dequeue_iq(iq, iq', \uparrow (TCP seg)) \wedge
 sock = SOCK(\uparrow fid, sf, \uparrow i₁, \uparrow p₁, \uparrow i₂, \uparrow p₂, es, cantsndmore, cantrcvmore,
 TCP_Sock(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge
 st \notin {CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} \wedge

(\exists seq_discard ack_discard URG_discard ACK_discard PSH_discard SYN_discard FIN_discard
 win_discard ws_discard urp_discard mss_discard ts_discard data_discard.

seg = \langle
 is₁ := \uparrow i₂;
 is₂ := \uparrow i₁;
 ps₁ := \uparrow p₂;
 ps₂ := \uparrow p₁;
 seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign);
 ack := tcp_seq_flip_sense(ack_discard : tcp_seq_local);
 URG := URG_discard;
 ACK := ACK_discard;
 PSH := PSH_discard;
 RST := **T**;
 SYN := SYN_discard;
 FIN := FIN_discard;
 win := win_discard;
 ws := ws_discard;
 urp := urp_discard;
 mss := mss_discard;
 ts := ts_discard;
 data := data_discard
 \rangle
 $\rangle \wedge$

((* sock.st \in {CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} excluded already above *)
if st \in {ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSE_WAIT} **then**
 err = \uparrow ECONNRESET
else (* sock.st \in {CLOSING; LAST_ACK} – leave existing error *)
 err = sock.es) \wedge

(* see tcp_close (p121) *)
 sock' = tcp_close h.arch(sock \langle es := err \rangle)

deliver_in_7a **tcp: network nonurgent** Receive RST and zap SYN_RECEIVED socket

$$h \langle \langle socks := socks \oplus [(sid, sock)]; \quad \xrightarrow{\tau} \quad h \langle \langle socks := socks \oplus socks_update'; \\ iq := iq \rangle \rangle \quad \quad \quad iq := iq' \rangle \rangle$$

(* **Summary:** receive RST and zap SYN_RECEIVED socket, removing from listen queue etc. *)

dequeue_iq(iq, iq', \uparrow (TCP seg)) \wedge

($\exists seq_discard \ ack_discard \ URG_discard \ ACK_discard \ PSH_discard \ SYN_discard \ FIN_discard$
 $win_discard \ ws_discard \ urp_discard \ mss_discard \ ts_discard \ data_discard$.)

$$seg = \langle \langle \begin{array}{l} is_1 := \uparrow i_2; \\ is_2 := \uparrow i_1; \\ ps_1 := \uparrow p_2; \\ ps_2 := \uparrow p_1; \\ seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign); \\ ack := tcp_seq_flip_sense(ack_discard : tcp_seq_local); \\ URG := URG_discard; \\ ACK := ACK_discard; \\ PSH := PSH_discard; \\ RST := \mathbf{T}; \\ SYN := SYN_discard; \\ FIN := FIN_discard; \\ win := win_discard; \\ ws := ws_discard; \\ urp := urp_discard; \\ mss := mss_discard; \\ ts := ts_discard; \\ data := data_discard \end{array} \rangle \rangle$$

) \wedge

$sid \notin \mathbf{dom}(socks) \wedge$

$$sock = \mathbf{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \\ \mathbf{TCP_Sock}(\mathbf{SYN_RECEIVED}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$$

((* There is a corresponding listening socket – passive open *)

($\exists (sid', lsock) :: socks \setminus sid$.)

$\exists tcp_lsock \ lis \ q0L \ q0R \ lsock'$

$lsock.pr = \mathbf{TCP_PROTO}(tcp_lsock) \wedge$

$tcp_lsock.st = \mathbf{LISTEN} \wedge$

$tcp_lsock.lis = \uparrow lis \wedge$

$lis.q_0 = q0L @ (sid :: q0R) \wedge$

$lsock' = lsock$

$\langle \langle pr := \mathbf{TCP_PROTO}(tcp_lsock \langle \langle lis := \\ \uparrow (lis \langle \langle q_0 := q0L @ q0R \rangle \rangle \rangle \rangle) \rangle \rangle \rangle \wedge$

$socks_update' = [(sid', lsock'); (sid, sock')]$

) \vee

((* No corresponding socket – simultaneous open *)

$socks_update' = [(sid, sock')] \wedge$

(* We do not delete the socket entry here because of simultaneous opens. Keep existing error for SYN_RECEIVED socket on RST *)
 $sock' = (tcp_close\ h.arch\ sock)\langle ps_1 := \mathbf{if}\ bsd_arch\ h.arch\ \mathbf{then}\ *\ \mathbf{else}\ sock.ps_1 \rangle$

deliver_in_7b **tcp: network nonurgent Receive RST and ignore for LISTEN socket**

$h\langle socks := socks \oplus [(sid, sock)];\ iq := iq \rangle \xrightarrow{\tau} h\langle socks := socks \oplus [(sid, sock)];\ iq := iq' \rangle$

(* **Summary:** receive RST and ignore for LISTEN socket *)

$dequeue_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$
 $sock = SOCK(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
 $TCP_Sock(LISTEN, cb, lis, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

(* BSD listen bug – since we can call listen() from any state, the peer IP/port may have been set *)
 $((is_2 = * \wedge ps_2 = *) \vee$
 $(bsd_arch\ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

$i_1 \in local_ips\ h.ifds \wedge$
 $\mathbf{T} \wedge$ (* placeholder for not a link-layer multicast or broadcast *)
 (* seems unlikely, since $i_1 \in local_ips\ h.ifds$ *)
 $\neg(is_broadmcast\ h.ifds\ i_1) \wedge$
 $\neg(is_broadmcast\ h.ifds\ i_2) \wedge$
(case is_1 **of**
 $\uparrow i1' \rightarrow i1' = i_1 \parallel$
 $* \rightarrow \mathbf{T}) \wedge$

$(\exists seq_discard\ ack_discard\ URG_discard\ ACK_discard\ PSH_discard\ SYN_discard\ FIN_discard$
 $win_discard\ ws_discard\ urp_discard\ mss_discard\ ts_discard\ data_discard.$

$seg = \langle$
 $is_1 := \uparrow i_2;$
 $is_2 := \uparrow i_1;$
 $ps_1 := \uparrow p_2;$
 $ps_2 := \uparrow p_1;$
 $seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign);$
 $ack := tcp_seq_flip_sense(ack_discard : tcp_seq_local);$
 $URG := URG_discard;$
 $ACK := ACK_discard;$
 $PSH := PSH_discard;$
 $RST := \mathbf{T};$
 $SYN := SYN_discard;$
 $FIN := FIN_discard;$
 $win := win_discard;$
 $ws := ws_discard;$
 $urp := urp_discard;$
 $mss := mss_discard;$
 $ts := ts_discard;$
 $data := data_discard$
 \rangle
 $\rangle \wedge$

$tcp_socket_best_match(socks \setminus sid)(sid, sock)seg\ h.arch$ (* there does not exist a better socket match to which the segment should be sent *)

deliver_in_7c **tcp: network nonurgent** Receive RST and ignore for SYN_SENT(unacceptable ack) or TIME_WAIT socket

$$h \langle \text{socks} := \text{socks} \oplus [(sid, sock)]; \quad \tau \rightarrow \quad h \langle \text{socks} := \text{socks} \oplus [(sid, sock')]; \\ iq := iq \rangle \quad \quad \quad iq := iq' \rangle$$

(* **Summary:** receive RST and ignore for SYN_SENT(unacceptable ack) or TIME_WAIT socket *)

dequeue_iq(iq, iq', ↑(TCP seg)) ∧
 sid ∉ **dom**(socks) ∧
 sock = SOCK(↑ fid, sf, ↑ i₁, ↑ p₁, ↑ i₂, ↑ p₂, es, cantsndmore, cantrcvmore,
 TCP_Sock(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) ∧
 st ∈ {SYN_SENT; TIME_WAIT} ∧

(∃ seq_discard URG_discard PSH_discard SYN_discard FIN_discard
 win_discard ws_discard urp_discard mss_discard ts_discard data_discard.

seg = ⟨
 is₁ := ↑ i₂;
 is₂ := ↑ i₁;
 ps₁ := ↑ p₂;
 ps₂ := ↑ p₁;
 seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign);
 ack := tcp_seq_flip_sense(ack : tcp_seq_local);
 URG := URG_discard;
 ACK := ACK;
 PSH := PSH_discard;
 RST := **T**;
 SYN := SYN_discard;
 FIN := FIN_discard;
 win := win_discard;
 ws := ws_discard;
 urp := urp_discard;
 mss := mss_discard;
 ts := ts_discard;
 data := data_discard
 ⟩
) ∧

(* no- or unacceptable- ACK *)

(st = SYN_SENT ⇒
 (¬ACK ∨ (ACK ∧ ¬(cb.iss < ack ∧ ack ≤ cb.snd_max)))) ∧

sock.pr = TCP_PROTO(tcp_sock) ∧

(if st = TIME_WAIT then (* only update if ≥ ESTABLISHED, c.f. tcp_input.c:887 *)

sock' = sock ⟨ pr := TCP_PROTO(tcp_sock)
 ⟨ cb := cb
 ⟨ t_idletime := stopwatch_zero; (* just received segment *)
 tt_keep := ↑((())_{slow_timer} TCPTV_KEEP_IDLE) ⟩ ⟩

else (* st = SYN_SENT *)

(* BSD rcv_wnd bug: the receive window updated code in tcp_input gets executed before the segment is processed, so even for bad segments, it gets updated *)

let rcv_window = calculate_bsd_rcv_wnd sf tcp_sock in

sock' = sock ⟨ pr := TCP_PROTO(tcp_sock)
 ⟨ cb := cb

```

    ⟨ rcv_wnd := if bsd_arch h.arch then rcv_window else tcp_sock.cb.rcv_wnd;
      rcv_adv := if bsd_arch h.arch then tcp_sock.cb.rcv_nxt + rcv_window
                else tcp_sock.cb.rcv_adv
    ⟩
  ⟩
)
)
)

```

deliver_in_7d **tcp: network nonurgent** Receive RST and zap SYN_SENT(acceptable ack) socket

```

h ⟨ socks := socks ⊕ [(sid, sock)];   T → h ⟨ socks := socks ⊕ [(sid, sock')];
iq := iq ⟩                             iq := iq' ⟩

```

(* **Summary** Receiving an acceptable-ack RST segment: kill the connection and set the socket's error field appropriately, unless we are WinXP where we simply ignore the RST. *)

```

dequeue_iq(iq, iq', ↑(TCP seg)) ∧
sid ∉ dom(socks) ∧
sock = SOCK(↑ fid, sf, ↑ i1, ↑ p1, ↑ i2, ↑ p2, es, cantsndmore, cantrcvmore,
            TCP_Sock(SYN_SENT, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) ∧

```

```

(∃ seq_discard URG_discard PSH_discard SYN_discard FIN_discard
 win_discard ws_discard urp_discard mss_discard ts_discard data_discard.

```

```

seg = ⟨
  is1 := ↑ i2;
  is2 := ↑ i1;
  ps1 := ↑ p2;
  ps2 := ↑ p1;
  seq := tcp_seq_flip_sense(seq_discard : tcp_seq_foreign);
  ack := tcp_seq_flip_sense(ack : tcp_seq_local);
  URG := URG_discard;
  ACK := T;
  PSH := PSH_discard;
  RST := T;
  SYN := SYN_discard;
  FIN := FIN_discard;
  win := win_discard;
  ws := ws_discard;
  urp := urp_discard;
  mss := mss_discard;
  ts := ts_discard;
  data := data_discard
  ⟩
) ∧

```

```

cb.iss < ack ∧ ack ≤ cb.snd_max ∧ (* acceptable ack *)

```

```

(if windows_arch h.arch then

```

```

  sock' = sock (* Windows XP just ignores RST's with a valid ack during connection establishment *)

```

```

else

```

```

  (∃ err.

```

```

    err ∈ {ECONNREFUSED; ECONNRESET} ∧ (* Note it is unclear whether or not this error will overwrite
                                         any existing error on the socket *)

```

```

    sock' = (tcp_close h.arch sock) ⟨ ps1 := if bsd_arch h.arch then * else sock.ps1;
                                     es := ↑ err ⟩
  )
)

```

deliver_in_8 **tcp: network nonurgent** Receive SYN in non-{CLOSED; LISTEN; SYN_SENT; TIME_WAIT} state

$$\begin{array}{l}
 h \langle \text{socks} := \text{socks} \oplus [(sid, sock)]; \quad \xrightarrow{T} \quad h \langle \text{socks} := \text{socks} \oplus [(sid, sock)']; \\
 iq := iq; \quad \quad \quad iq := iq'; \\
 oq := oq; \quad \quad \quad oq := oq'; \\
 bndlm := bndlm \rangle \quad \quad \quad bndlm := bndlm' \rangle
 \end{array}$$

(* **Summary:** Receive a SYN in non-{CLOSED; LISTEN; SYN_SENT; TIME_WAIT} state. Drop it and (depending on the architecture) generate a RST. *)

```

dequeue_iq(iq, iq', ↑(TCP seg)) ∧
sid ∉ dom(socks) ∧
sock = SOCK(↑ fid, sf, ↑ i1, ↑ p1, ↑ i2, ↑ p2, es, cantndmore, cantrcvmore,
            TCP_Sock(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) ∧
(∃ ws_discard mss_discard.
seg = ⟨
  is1 := ↑ i2;
  is2 := ↑ i1;
  ps1 := ↑ p2;
  ps2 := ↑ p1;
  seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);
  ack := tcp_seq_flip_sense(ack : tcp_seq_local);
  URG := URG;
  ACK := ACK;
  PSH := PSH;
  RST := F;
  SYN := T;
  FIN := FIN;
  win := win;
  ws := ws_discard;
  urp := urp;
  mss := mss_discard;
  ts := ts;
  data := data
⟩
) ∧

```

(* Note that it may be the case that this rule should only apply when the SYN is *in the trimmed window*, should not it?; it's OK if there's a SYN bit set, for example in a retransmission. *)

$st \notin \{\text{CLOSED}; \text{LISTEN}; \text{SYN_SENT}; \text{TIME_WAIT}\} \wedge$

```

sock.pr = TCP_PROTO(tcp_sock) ∧
let t_idletime' = stopwatch_zero in
let tt_keep' = if tcp_sock.st ≠ SYN_RECEIVED then
  ↑((( ))_slow_timer TCPTV_KEEP_IDLE)
else
  tcp_sock.cb.tt_keep in
let tt_fin_wait_2' = if tcp_sock.st = FIN_WAIT_2 then
  ↑((( ))_slow_timer TCPTV_MAXIDLE)
else
  tcp_sock.cb.tt_fin_wait_2 in

```

```

sock' = sock ⟨ pr := TCP_PROTO(tcp_sock
  ⟨ cb := tcp_sock.cb ⟨ tt_keep := tt_keep'

```


$$tt_fin_wait_2 := tt_fin_wait_2';$$

$$t_idletime := t_idletime'$$

$$\rangle\rangle\rangle \wedge$$

(if `bsd_arch h.arch` then `make_rst_segment_from_cb tcp_sock.cb(i1, i2, p1, p2) seg'` else **T**) \wedge
`dropwithreset seg h.ifds(ticks_of h.ticks) BANDLIM_UNLIMITED bndlm bndlm' outsegs` \wedge
`outsegs' = (if bsd_arch h.arch then (TCP(seg')) :: outsegs else outsegs)` \wedge
`enqueue_each_and_ignore_fail h.arch h.rttab h.ifds outsegs' oq oq'`

deliver_in_9 **tcp: network nonurgent** Receive SYN in TIME_WAIT state if there is no matching LISTEN socket or sequence number has not increased

$$h \langle\langle socks := socks \oplus [(sid, sock)]; \quad \xrightarrow{\tau} \quad h \langle\langle socks := socks \oplus [(sid, sock)];$$

$$iq := iq; \quad \quad \quad iq := iq';$$

$$oq := oq; \quad \quad \quad oq := oq';$$

$$bndlm := bndlm \rangle \rangle \quad \quad \quad bndlm := bndlm' \rangle \rangle$$

(* **Summary:** Receive a SYN in TIME_WAIT} state where there is no matching LISTEN socket. Drop it and (depending on the architecture) generate a RST. *)

$$\text{dequeue_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$$

$$sid \notin \mathbf{dom}(socks) \wedge$$

$$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$$

$$\text{TCP_Sock}(\text{TIME_WAIT}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$$

$$(\exists ws_discard mss_discard.$$

$$seg = \langle\langle$$

$$is_1 := \uparrow i_2;$$

$$is_2 := \uparrow i_1;$$

$$ps_1 := \uparrow p_2;$$

$$ps_2 := \uparrow p_1;$$

$$seq := tcp_seq_flip_sense(seq : tcp_seq_foreign);$$

$$ack := tcp_seq_flip_sense(ack : tcp_seq_local);$$

$$URG := URG;$$

$$ACK := ACK;$$

$$PSH := PSH;$$

$$RST := \mathbf{F};$$

$$SYN := \mathbf{T};$$

$$FIN := FIN;$$

$$win := win;$$

$$ws := ws_discard;$$

$$urp := urp;$$

$$mss := mss_discard;$$

$$ts := ts;$$

$$data := data$$

$$\rangle \rangle$$

$$\rangle \wedge$$

(* no matching LISTEN socket, or the sequence number has not increased *)

$$((seq \leq (tcp_sock_of sock).cb.rcv_next)$$

$$\vee$$

$$\neg(\exists((sid, sock) :: socks) tcp_sock.$$

$$sock.pr = \text{TCP_PROTO}(tcp_sock) \wedge$$

$$tcp_sock.st = \text{LISTEN} \wedge$$

```

  sock.is1 ∈ {*; ↑ i1} ∧
  sock.ps1 = ↑ p1)
) ∧

```

```

(if bsd_arch h.arch then make_rst_segment_from_cb cb(i1, i2, p1, p2)seg' else T) ∧
dropwithreset seg h.ifds(ticks_of h.ticks)BANDLIM_RST_CLOSEDPORT bndlm bndlm' outsegs ∧
outsegs' = (if bsd_arch h.arch then (TCP(seg')) :: outsegs else outsegs) ∧
enqueue_each_and_ignore_fail h.arch h.rttab h.ifds outsegs' oq oq'

```

(* This rule does not appear in the BSD code; what happens there is that the old TIME_WAIT state socket is closed, and then the code jumps back to the top. So this rule covers the case where it then discovers nothing else is listening, like *deliver_in_5*. *)

Chapter 17

Host LTS: TCP Output

17.1 Output (TCP only)

A TCP implementation would typically perform output deterministically, e.g. during the processing a received segment it may construct and enqueue an acknowledgement segment to be emitted. This means that the detailed behaviour of a particular implementation depends on exactly where the output routines are called, affecting when segments are emitted. The contents of an emitted segment, on the other hand, must usually be determined by the socket state (especially the `tcpcb`), not from transient program variables, so that retransmissions can be performed.

In this specification we choose to be somewhat nondeterministic, loosely specifying when common-case TCP output to occur. This simplifies the modelling of existing implementations (avoiding the need to capture the code points at which the output routines are called) and should mean the specification is closer to capturing the set of all reasonable implementations.

A significant defect in the current specification is that it does not impose a very tight lower bound on how often output takes place. The satisfactory dynamic behaviour of TCP connections depends on an "ACK clock" property, with receivers acknowledging data sufficiently often to update the sender's send window. Characterising this may need additional constraints.

The rule presented in this chapter describes TCP output in the common case, i.e. the behaviour of TCP when emitting a non-SYN, non-RST segment. The whole behaviour is captured by the single rule `deliver_out_1` which relies upon the auxiliary functions `tcp_output_required` (p111) and `tcp_output_really` (p113). Output (strictly, adding segments to the host's output queue) may take place whenever this rule can fire; it does construct the output segments purely from the socket state.

The two auxiliary functions are loosely based on BSD's TCP output function, which can be logically divided into two halves. The first of these —to some approximation— is a guard that prevents output from occurring unless it is valid to do so, and the second actually creates a segment and passes it to the IP layer for output. This distinction is mirrored in the specification, with `tcp_output_required` acting as the guard and `tcp_output_really` forming the segment ready to be appended to the host's output queue. Unfortunately it is not possible to be as clean here as one might hope, because under some circumstances `tcp_output_required` may have side-effects. It should be noted that `tcp_output_really` only creates a segment and does not perform any "output" — the act of adding the segment (perhaps unreliably) to the host's output queue is the job of the caller.

The output cases not covered by `deliver_out_1` are handled specially and often in a more deterministic way. Segments with the SYN flag set are created by the auxiliary functions `make_syn_segment` (p106) and `make_syn_ack_segment` (p107) and are output deterministically in response to either user events or segment input. SYN segments are emitted by the rules commonly involved in connection establishment, namely `connect_1`, `deliver_in_1`, `deliver_in_2`, `timer_tt_rexmtsyn_1` and `timer_tt_rexmt_1` and are special-cased in this way for clarity because connection establishment performs extra work such as option negotiation and state initialisation.

The creation of RST segments is performed by the auxiliaries `make_rst_segment_from_cb` (p109) and `make_rst_segment_from_seg` (p110), and are used by the rules that require a reset segment to be emitted in response to a user event, e.g. a `close()` call on a socket with a zero linger time, or as a socket's response to receiving some types of invalid segment.

In a few places, mainly in the specification of certain congestion control methods, some rules use `tcp_output_really` (p113) or the wrapper functions `tcp_output_perhaps` (p116) and

mlift_tcp_output_perhaps_or_fail (p118) directly and—more importantly—deterministically. This is partly for clarity, perhaps because an RFC states that output "MUST" occur at that point, and partly for convenience, possibly because the model would require much extra state (hence adding unnecessary complexity) if the output function was not used in-place.

The tcp_output_perhaps function almost entirely mimics an implementation's TCP output function. It calls tcp_output_required to check that output can take place, applying any side-effects that it returns, and finally creates the segment with tcp_output_really. See tcp_output_perhaps (p116) and mlift_tcp_output_perhaps_or_fail (p118) for more information.

Other auxiliary functions are involved in TCP output and are described earlier. Once a segment has been constructed it is added to the host's output queue by one of enqueue_or_fail (p118), enqueue_or_fail_sock (p118), enqueue_and_ignore_fail (p118), enqueue_each_and_ignore_fail (p118) or mlift_tcp_output_perhaps_or_fail (p118). These functions are used by *deliver_out_1* and other rules in the specification to non-deterministically add a segment to the host's output queue. In the common case, a segment is added to the host's output queue successfully. In other cases, the auxiliary function rollback_tcp_output (p117) may assert a segment is unroutable and prevent the segment from being added to the queue. Some failures are non-deterministic in order to model "out of resource" style errors, although most are deterministic routing failures determined from the socket and host states. rollback_tcp_output has a second task to "undo" several of the socket's control block changes upon an error condition. Some of the enqueue functions ignore failure, e.g. enqueue_and_ignore_fail, and upon an error they just fail to queue the segment and do not update the socket with the "rolled-back" control block returned by rollback_tcp_output.

17.1.1 Summary

deliver_out_1 **tcp: network nonurgent** Common case TCP output

17.1.2 Rules

deliver_out_1 **tcp: network nonurgent** Common case TCP output

$$h \langle \langle socks := socks \oplus [(sid, sock)]; \quad \tau \rightarrow \quad h \langle \langle socks := socks \oplus [(sid, sock'')]; \\ oq := oq \rangle \rangle \quad \quad \quad oq := oq' \rangle$$

(* **Summary:** output TCP segment if possible. In some cases update the socket's persist timer without performing output. *)

(* The TCP socket is connected *)

$sid \notin \mathbf{dom}(socks) \wedge$
 $sock = \text{SOCK}(fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore,$
 $\quad \quad \quad cantrcvmore, \text{TCP_PROTO}(tcp_sock)) \wedge$
 $tcp_sock = \text{TCP_Sock0}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc) \wedge$

(* and either is in a synchronised state with initial SYN acknowledged... *)

$((st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}; \text{FIN_WAIT_1}; \text{FIN_WAIT_2}; \text{CLOSING};$
 $\quad \quad \quad \text{LAST_ACK}; \text{TIME_WAIT}\} \wedge$

$cb.snd_una \neq cb.iss) \vee$

(* ...or is in the SYN_SENT or SYN_RECEIVED state and a FIN needs to be emitted *)

$(st \in \{\text{SYN_SENT}; \text{SYN_RECEIVED}\} \wedge cantsndmore \wedge cb.tf_shouldacknow)$
 $) \wedge$

(* A segment will be emitted if tcp_output_required asserts that a segment can be output (*do_output*). If tcp_output_required returns a function to alter the socket's persist timer (*persist_fun*), then this does not of itself mean that a segment is required, however *deliver_out_1* should still fire to allow the update to take place. *)

let (*do_output*, *persist_fun*) = tcp_output_required *h.arch h.ifds sock* **in**
 $(do_output \vee persist_fun \neq *) \wedge$

(* Apply any persist timer side-effect from tcp_output_required *)

```

let sock0 = option_case sock(λf.sock
    ⟨ pr := TCP_PROTO(tcp_sock cb := f) ⟩)persist_fun in

(if do_output then (* output a segment *)
  (* Construct the segment to emit, updating the socket's state *)
  tcp_output_really h.arch F(ticks_of h.ticks)h.ifds sock0(sock', outsegs') ∧

  sock'.pr = TCP_PROTO(tcp_sock') ∧

  (* Add the segment to the host's output queue, rolling back the socket's control block state if an error occurs *)
  enqueue_or_fail_sock(tcp_sock'.st ∈ {CLOSED; LISTEN; SYN_SENT})h.arch h.rtttab h.ifds
    outsegs' oq sock0 sock'(sock'', oq')

else (* Do not output a segment, but ensure things are tidied up *)
  oq = oq' ∧
  sock'' = sock0
)

```

Chapter 18

Host LTS: TCP Timers

18.1 Timers (TCP only)

18.1.1 Summary

| | | |
|----------------------------|-------------------------------|--|
| <i>timer_tt_rexmtsyn</i> | tcp: misc nonurgent | SYN retransmit timer expires |
| <i>timer_tt_rexmt_1</i> | tcp: misc nonurgent | retransmit timer expires |
| <i>timer_tt_persist_1</i> | tcp: misc nonurgent | persist timer expires |
| <i>timer_tt_keep_1</i> | tcp: network nonurgent | keepalive timer expires |
| <i>timer_tt_2msl_1</i> | tcp: misc nonurgent | 2*MSL timer expires |
| <i>timer_tt_delack_1</i> | tcp: misc nonurgent | delayed-ACK timer expires |
| <i>timer_tt_conn_est</i> | tcp: misc nonurgent | connection establishment timer expires |
| <i>timer_tt_fin_wait_2</i> | tcp: misc nonurgent | FIN_WAIT_2 timer expires |

18.1.2 Rules

timer_tt_rexmtsyn_1 **tcp: misc nonurgent** SYN retransmit timer expires

$h \langle \langle socks := socks \oplus [(sid, sock)]; \quad \tau \rightarrow \quad h \langle \langle socks := socks \oplus [(sid, sock')];$
 $oq := oq \rangle \rangle \quad \quad \quad oq := oq' \rangle \rangle$

$sock.pr = TCP_PROTO(tcp_sock) \wedge$
 $tcp_sock.cb.tt_rexmt = \uparrow(((REXMTSYN, shift))_d) \wedge$
 $timer_expires \ d \wedge (* \text{ timer has expired } *)$

$tcp_sock.st = SYN_SENT \wedge (* \text{ this rule is incomplete: REXMTSYN is possible in other states, since } deliver_in_2 \text{ may}$
 $\text{change state without clearing } tt_rexmt \text{ } *)$

$cb = tcp_sock.cb \wedge$

(if $shift + 1 \geq TCP_MAXRXTSHIFT$ **then**

(* Timer has expired too many times. Drop and close the connection *)

(* since socket state is SYN_SENT, no segments can be output *)

$tcp_drop_and_close \ h.arch(\uparrow ETIMEDOUT)sock(sock', []) \wedge$
 $oq' = oq$

else

(* Update the control block based upon the number of occasions on which the timer expired *)

(if $shift + 1 = 1 \wedge cb.t_rttinf.tf_srtt_valid$ **then** (* On the first retransmit store values for recovery from a bad retransmit *)

(* we cannot guess the safe window for this if we do not know the RTT, hence the second condition *)

```

    snd_cwnd_prev' = cb.snd_cwnd ∧
    snd_ssthresh_prev' = cb.snd_ssthresh ∧
    t_badrxtwin' = ((TIMEWINDOW)kern_timer(time(cb.t_rttinf.t_srtt/2)) (* kern_timer for a ticks-based deadline *))
else (* Otherwise keep the previous values *)
    snd_cwnd_prev' = cb.snd_cwnd_prev ∧
    snd_ssthresh_prev' = cb.snd_ssthresh_prev ∧
    t_badrxtwin' = cb.t_badrxtwin (* should be TimeWindowClosed, since retransmit timer is always longer than
                                   t_srtt/2 *)
) ∧

(if (shift + 1 = 3) ∧ ¬(linux_arch h.arch) then (* On the third retransmit turn off window scaling and times-
                                                tamping options *))

    tf_req_tstamp' = F ∧
    request_r_scale' = *
else (* Otherwise keep the previous values *)
    tf_req_tstamp' = cb.tf_req_tstamp ∧
    request_r_scale' = cb.request_r_scale
) ∧

let t_rttinf' =
  (if shift + 1 > TCP_MAXRXTSHIFT div 4 then
    (* Invalidate the recorded smoothed round-trip time for the connection after TCP_MAXRXTSHIFT div 4
       retransmits *)
    (* Note that the BSD code adjusts the srtt and rttvar values here to ensure that if it does not get a new rtt
       measurement before the next retransmit it can still use the existing values. We do not need to do this for two
       reasons: (1) we have a flag to invalidate the srtt values (the only reason BSD updates srtt to be zero and hacks
       rttvar is to mark it invalid and request a new rtt update), and (2) the BSD_RTTVAR_BUG does not affect
       SYN retransmits in any case (because for SYN retransmits srtt is zero and BSD hacks up rttvar appropriately
       at the start of a new connection to make everything just work) *)
    (* Note that the socket's route should be discarded. *)
    cb.t_rttinf ⟨ tf_srtt_valid := F ⟩
  else
    cb.t_rttinf) in

cb' = cb ⟨ (* Restart the rexmt timer to time the retransmitted SYN *)
          tt_rexmt := start_tt_rexmtsyn h.arch(shift + 1)F cb.t_rttinf;
          (* reset to next backoff point *)
          t_badrxtwin := t_badrxtwin';
          t_rttinf := t_rttinf'
          ⟨ t_lastshift := shift + 1;
            t_wassyn := T ⟩;
          tf_req_tstamp := tf_req_tstamp';
          request_r_scale := request_r_scale';
          snd_next := cb.iss + 1; (* value after sending SYN *)
          snd_recover := cb.iss + 1; (* value after sending SYN *)

          t_rttseg := *;
          snd_cwnd := cb.t_maxseg;
          (* Calculation as per BSD *)
          snd_ssthresh := cb.t_maxseg * max 2(min cb.snd_wnd cb.snd_cwnd
                                             div(2 * cb.t_maxseg));

          snd_cwnd_prev := snd_cwnd_prev';
          snd_ssthresh_prev := snd_ssthresh_prev';
          t_dupacks := 0) ∧

(∃i1 i2 p1 p2.(sock.is1, sock.is2, sock.ps1, sock.ps2) = (↑ i1, ↑ i2, ↑ p1, ↑ p2) ∧

  (* Create the segment to be retransmitted *)
  choose seg' :: (make_syn_segment cb'(i1, i2, p1, p2)(ticks_of h.ticks)).

```

```

(* Attempt to add the new segment to the host's output queue, constraining the final control block state *)
enqueue_or_fail F h.arch h.rttab h.ifds[TCP seg'] oq
    (cb ⟨ snd_nxt := cb.iss; tt_delack := *;
        last_ack_sent := tcp_seq_foreign 0w; rcv_adv := tcp_seq_foreign 0w
        ⟩) cb'(cb'', oq')
) ∧
sock' = sock ⟨ pr := TCP_PROTO(tcp_sock ⟨ cb := cb'' ⟩) ⟩
)

```

timer_tt_rexmt_1 **tcp: misc nonurgent** retransmit timer expires

```

h ⟨ socks := socks ⊕     $\xrightarrow{\tau}$     h ⟨ socks := socks ⊕
  [(sid, sock)];                    [(sid, sock'')];
oq := oq ⟩                            oq := oq' ⟩

```

```

sock.pr = TCP_PROTO(tcp_sock) ∧
sock'.pr = TCP_PROTO(tcp_sock') ∧
(tcp_sock.st ∉ {CLOSED; LISTEN; SYN_SENT; CLOSE_WAIT; FIN_WAIT_2; TIME_WAIT}) ∨
(tcp_sock.st = LISTEN ∧ bsd_arch h.arch) ∧

```

```

tcp_sock.cb.tt_rexmt = ↑(((REXMT, shift))d) ∧
timer_expires d ∧

```

```

cb = tcp_sock.cb ∧

```

```

(if shift + 1 > (if tcp_sock.st = SYN_RECEIVED then TCP_SYNACKMAXRXTSHIFT
                  else TCP_MAXRXTSHIFT) then

```

```

  (* Note that BSD's syncaches have a much lower threshold for retransmitting SYN,ACKs than normal *)
  (* drop connection *)

```

```

tcp_drop_and_close h.arch(↑ ETIMEDOUT)sock(sock', [TCP seg']) (* will always get exactly one segment *)

```

else

```

  (* on first retransmit, store values for recovery from bad retransmit *)
  (* we cannot guess the safe window for this if we do not know the RTT, hence the second condition *)

```

```

(if shift + 1 = 1 ∧ cb.t_rttinf.tf_srtt_valid then
  snd_cwnd_prev' = cb.snd_cwnd ∧
  snd_ssthresh_prev' = cb.snd_ssthresh ∧
  t_badrxtwin' = ((TIMEWINDOW)kern_timer(time(cb.t_rttinf.t_srtt/2))) (* kern_timer for a ticks-based deadline *)

```

else

```

  snd_cwnd_prev' = cb.snd_cwnd_prev ∧
  snd_ssthresh_prev' = cb.snd_ssthresh_prev ∧
  t_badrxtwin' = cb.t_badrxtwin) ∧ (* should be TimeWindowClosed, since retransmit timer is always longer
                                    than t_srtt/2 *)

```

```

(* NB: The socket is not in SYN_SENT here; the rexmt timer has been split into two, and SYN_SENT uses
tt_rexmtsyzn. *)

```

```

let t_rttinf' = (if shift + 1 > TCP_MAXRXTSHIFT div 4 then
  (* Note that the socket's route should be discarded. *)

```

```

  cb.t_rttinf ⟨
    tf_srtt_valid := F;
    t_srtt := (cb.t_rttinf.t_srtt/4)
    onlywhen(bsd_arch h.arch ∧ BSD_RTTVAR_BUG)
  ⟩

```



```

    else
      cb.t_rttnf) in

(* backoff the timer and do a retransmit *)
cb' = cb ( tt_rexmt := start_tt_rexmt h.arch(shift + 1) F cb.t_rttnf; (* reset to next backoff point *)
  (* tcp_output_really touches this again, but actually leaves it the same, unless sock.snd_urp is set and
  win0 ≠ 0, weirdly *)
  t_badrxtwin := t_badrxtwin';
  t_rttnf := t_rttnf' (
    t_lastshift := shift + 1;
    t_wassyn := F
  );
  snd_nxt := cb.snd_una; (* want to retransmit from snd_una *)
  snd_recover := cb.snd_max;
  t_rttnf := *;
  snd_cwnd := cb.t_maxseg;
  snd_ssthresh := cb.t_maxseg * max 2(min cb.snd_wnd cb.snd_cwnd div(2 * cb.t_maxseg));
  snd_cwnd_prev := snd_cwnd_prev';
  snd_ssthresh_prev := snd_ssthresh_prev';
  t_dupacks := 0) ∧

(if tcp_sock.st = SYN_RECEIVED then
  (∃ i1 i2 p1 p2.

  (* If we're Linux doing a simultaneous open and support timestamping then ensure timestamping is enabled
  in any retransmitted SYN,ACK segments. See deliver_in_2 for the rationale in full, but in short Linux is
  RFC1323 compliant and makes a hash of option negotiation during a simultaneous open. We make the option
  decision early (as per the RFC and BSD) and have to hack up SYN,ACK segments to contain timestamp
  options if the Linux host supports timestamping. *)
  (* Note: this behaviour is also safe if we are here due to a passive open. In this case, if the remote end
  does not support timestamping, tf_req_tstamp is F due to the option negotiation in deliver_in_1. Then
  tf_doing_tstamp is necessarily F too and the retransmitted SYN,ACK segment does not contain a timestamp.
  OTOH, if tf_req_tstamp is still T then so is tf_doing_tstamp and the faked up cb below is safe. *)
  (* Note that similar to the above note on timestamping, window scaling may also have to be dealt with
  here. *)
  let cb''' =
    (if ((linux_arch h.arch) ∧ cb.tf_req_tstamp) then
      cb' ( tf_req_tstamp := T;
        tf_doing_tstamp := T)
    else
      cb') in

  (* Note that tt_delack and possibly other timers should be cleared here *)
  (sock.is1, sock.is2, sock.ps1, sock.ps2) = (↑ i1, ↑ i2, ↑ p1, ↑ p2) ∧

  (* We are in SYN_RECEIVED and want to retransmit the SYN,ACK, so we either got here via deliver_in_1
  or deliver_in_2. In both cases, calculate_buf_sizes was used to set cb.t_maxseg to the correct value (as
  per tcp_mss() in BSD), however, we need to use the old values in retransmitting the SYN,ACK, as per
  tcp_mssopt() in BSD. make_syn_ack_segment therefore uses the value stored in cb.t_advms to set the same
  mss option in the segment, so we do not need to do anything special here. *)
  seg' ∈ make_syn_ack_segment cb'''(i1, i2, p1, p2)(ticks_of h.ticks) ∧

  (* We need to remember to add the length of the segment data (i.e. 1 for a SYN) back onto snd_nxt in the
  cb, since this is what tcp_output_really does for normal retransmits. If we do not do this, then we'll end up
  trying to send the first lot of data with a seq of iss, rather than iss + 1 *)
  sock' = sock ( pr := TCP_PROTO(tcp_sock ( cb := cb'
    ( snd_nxt := cb'.snd_nxt + 1)))
  )
)

```

```

else if tcp_sock.st = LISTEN then (* BSD LISTEN bug: in BSD it is possible to transition a socket to
the LISTEN state without cancelling the rexmt timer. In this case,
segments are emitted with no flags set. *)
  BSD_arch h.arch  $\wedge$ 
  ( $\exists i_1 i_2 p_1 p_2$ .
  (sock.is1, sock.is2, sock.ps1, sock.ps2) = ( $\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2$ )  $\wedge$ 
  seg'  $\in$  BSD_make_phantom_segment cb'(i1, i2, p1, p2)(ticks_of h.ticks)(sock.cantsndmore))  $\wedge$ 
  (* Retransmission only continues if FIN is set in the outgoing segment (really!) *)
  sock' = sock  $\llbracket$  pr := TCP_PROTO(tcp_sock
     $\llbracket$  cb := cb'  $\llbracket$  tt_rexmt := * onlywhen  $\neg$ seg'.FIN  $\rrbracket$   $\rrbracket$   $\rrbracket$ 
else (* ESTABLISHED, FIN_WAIT_1, CLOSING, LAST_ACK *)
  (* i.e., cannot be CLOSED, LISTEN, SYN_SENT, CLOSE_WAIT, FIN_WAIT_2, TIME_WAIT *)
  tcp_output_really h.arch F(ticks_of h.ticks)h.ifds
    (sock  $\llbracket$  pr := TCP_PROTO(tcp_sock  $\llbracket$  cb := cb'  $\rrbracket$   $\rrbracket$ )
    (sock', [TCP seg']) (* always emits exactly one segment *))
)
)  $\wedge$ 

enqueue_or_fail T h.arch h.rttab h.ifds [TCP seg'] oq
  cb' tcp_sock'.cb(cb'', oq')  $\wedge$ 
sock'' = sock'  $\llbracket$  pr := TCP_PROTO(tcp_sock'  $\llbracket$  cb := cb''  $\rrbracket$   $\rrbracket$ 

```

timer_tt_persist_1 **tcp: misc nonurgent persist timer expires**

```

h  $\llbracket$  socks := socks  $\oplus$   $\xrightarrow{\tau}$  h  $\llbracket$  socks := socks  $\oplus$ 
  [(sid, sock)]; [(sid, sock'')];
oq := oq  $\rrbracket$  oq := oq'  $\rrbracket$ 

```

```

sock.pr = TCP_PROTO(tcp_sock)  $\wedge$ 
sock'.pr = TCP_PROTO(tcp_sock')  $\wedge$ 
tcp_sock.cb.tt_rexmt =  $\uparrow$ ((PERSIST, shift)d)  $\wedge$ 
timer_expires d  $\wedge$ 
let sock0 = sock  $\llbracket$  pr := TCP_PROTO(tcp_sock
   $\llbracket$  cb := tcp_sock.cb
   $\llbracket$  tt_rexmt := start_tt_persist(shift + 1)tcp_sock.cb.t_rttinf h.arch  $\rrbracket$   $\rrbracket$  in

tcp_output_really h.arch T (* T indicates a window probe is requested *)
  (ticks_of h.ticks)h.ifds
  sock0
  (sock', outsegs')  $\wedge$ 
enqueue_or_fail_sock(tcp_sock'.st  $\in$  {CLOSED; LISTEN; SYN_SENT})h.arch h.rttab h.ifds
  outsegs' oq sock0 sock'(sock'', oq')

```

timer_tt_keep_1 **tcp: network nonurgent keepalive timer expires**

```

h  $\llbracket$  socks := socks  $\oplus$ 
  [(sid, SOCK( $\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$ 
    TCP_Sock(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]);
oq := oq  $\rrbracket$ 

```

$$\xrightarrow{\tau} h \llbracket socks := socks \oplus$$

$$\quad \left[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \text{cantsndmore}, \text{cantrcvmore}, \text{TCP_Sock}(st, cb', *, \text{sndq}, \text{sndurp}, \text{rcvq}, \text{rcvurp}, iobc))) \right];$$

$$oq := oq' \rrbracket$$

(* Note that in another rule the following needs to be specified: if the timer has expired for the last time, then (in another rule): (if HAVERCVDSYN (i.e., not CLOSED/LISTEN/SYN_SENT) then send a RST else do not do anything yet) \wedge copy soft error to es \wedge free tcpcb, saving RTT *)

$$cb.tt_keep = \uparrow(((())_d) \wedge$$

$$\text{timer_expires } d \wedge$$

(* Note the following condition also needs to be investigated: $cb.t_rcvtime + tcp_keepidle + tcp_keepcnt * tcp_keepintvl < NOW \wedge$ - still probing *)

$$(\exists win_.$$

$$\mathbf{w2n} \text{ win_} = cb.rcv_wnd \ggg cb.rcv_scale \wedge$$

let $ts = \mathbf{if}$ $cb.tf_doing_tstmp$ **then**

$$\quad \mathbf{let} \text{ } ts_ecr' = \mathbf{option_case} (ts_seq \ 0w) \ \mathbf{I} (\text{timewindow_val_of } cb.ts_recent) \ \mathbf{in}$$

$$\quad \uparrow(((\text{ticks_of } h.ticks), ts_ecr')$$

$$\mathbf{else}$$

$$\quad * \ \mathbf{in}$$

$$seg = \llbracket is_1 := \uparrow i_2;$$

$$is_2 := \uparrow i_1;$$

$$ps_1 := \uparrow p_2;$$

$$ps_2 := \uparrow p_1;$$

$$seg := cb.snd_una - 1; (* \text{deliberately outside window} *)$$

$$ack := cb.rcv_nxt;$$

$$URG := \mathbf{F};$$

$$ACK := \mathbf{T};$$

$$PSH := \mathbf{F};$$

$$RST := \mathbf{F};$$

$$SYN := \mathbf{F};$$

$$FIN := \mathbf{F};$$

$$win := win_;$$

$$ws := *;$$

$$urp := 0w;$$

$$mss := *;$$

$$ts := ts;$$

$$data := []$$

$$\rrbracket) \wedge$$

$$\text{enqueue_and_ignore_fail } h.arch \ h.rttab \ h.ifds[\text{TCP } seg] oq \ oq' \wedge$$

$$cb' = cb \llbracket tt_keep := \uparrow(((())_{\text{slow_timer}} \ \text{TCPTV_KEEPINTVL});$$

$$\quad last_ack_sent := seg.ack$$

$$\rrbracket$$

timer_tt_2msl_1 **tcp: misc nonurgent** **2*MSL timer expires**

$$h \llbracket socks := socks \oplus \quad \xrightarrow{\tau} \quad h \llbracket socks := socks \oplus$$

$$\quad \left[(sid, sock) \right] \rrbracket \quad \quad \quad \left[(sid, sock') \right] \rrbracket$$

(* **Summary:** When the 2MSL TIME_WAIT period expires, the socket is closed. *)

```

sock.pr = TCP_PROTO(tcp_sock) ∧
tcp_sock.cb.tt_2msl = ↑((( ))_d) ∧
timer_expires d ∧
sock' = tcp_close h.arch sock

```

timer_tt_delack_1 **tcp: misc nonurgent** **delayed-ACK timer expires**

```

h ⟨socks := socks ⊕   ↗   h ⟨socks := socks ⊕
  [(sid, sock)];      [(sid, sock'')];
oq := oq⟩             oq := oq'⟩

```

```

sock.pr = TCP_PROTO(tcp_sock) ∧
sock'.pr = TCP_PROTO(tcp_sock') ∧
tcp_sock.cb.tt_delack = ↑((( ))_d) ∧
timer_expires d ∧
let sock0 = sock ⟨pr := TCP_PROTO(tcp_sock) ⟨cb := tcp_sock.cb ⟨tt_delack := *⟩⟩⟩ in
tcp_output_really h.arch F(ticks_of h.ticks)h.ifds sock0(sock', outsegs') ∧
enqueue_or_fail_sock(tcp_sock'.st ∈ {CLOSED; LISTEN; SYN_SENT})h.arch h.rttab h.ifds
  outsegs' oq sock0 sock'(sock'', oq')

```

Description

This overlaps with *deliver_out_1*. This is a bit odd, but is a consequence of our liberal nondeterministic TCP output.

timer_tt_conn_est_1 **tcp: misc nonurgent** **connection establishment timer expires**

```

h ⟨socks := socks ⊕   ↗   h ⟨socks := socks ⊕
  [(sid, sock)];      [(sid, sock')];
oq := oq⟩             oq := oq'⟩

```

(* **Summary:** If the connection-establishment timer goes off, drop the connection (possibly *RST*ing the other end). *)

```

sock.pr = TCP_PROTO(tcp_sock) ∧
tcp_sock.cb.tt_conn_est = ↑((( ))_d) ∧
timer_expires d ∧
tcp_drop_and_close h.arch(↑ ETIMEDOUT)
  (sock ⟨pr := TCP_PROTO(tcp_sock) ⟨cb := tcp_sock.cb
    ⟨tt_conn_est := *⟩⟩⟩)(sock', outsegs) ∧

```

(* Note it should be the case that the socket is in SYN_SENT, and so *outsegs* will be empty, but that is not definite. *)
enqueue_and_ignore_fail h.arch h.rttab h.ifds outsegs oq oq'

Description

POSIX: says, in the *INFORMATIVE* section *APPLICATION USAGE*, that the state of the socket is unspecified if `connect()` fails. We could (in the POSIX "architecture") model this accurately.

timer_tt_fin_wait_2_1 **tcp: misc nonurgent** **FIN_WAIT_2 timer expires**

```

h ⟨socks := socks ⊕   ↗   h ⟨socks := socks ⊕
  [(sid, sock)⟩⟩      [(sid, sock')⟩⟩

```

```

sock.pr = TCP_PROTO(tcp_sock) ∧
tcp_sock.cb.tt_fin_wait_2 = ↑((( ))_d) ∧

```

```
timer_expires d ∧  
sock' = tcp_close h.arch sock
```

Description This stops the timer and closes the socket.

Unlike BSD, we take steps to ensure that this timer only fires when it is really time to close the socket. Specifically, we reset it every time we receive a segment while in FIN_WAIT_2, to TCPTV_MAXIDLE. This means we do not need any guarding conditions here; we just do it.

This means that we do not directly model the BSD behaviour of "sleep for 10 minutes, then check every 75 seconds to see if the connection has been idle for 10 minutes".

Chapter 19

Host LTS: UDP Input Processing

19.1 Input Processing (UDP only)

19.1.1 Summary

| | | | | |
|-------------------------|-------------|----------------|---------------|--|
| <i>deliver_in_udp_1</i> | udp: | network | nonur- | Get UDP datagram from host's in-queue and deliver it to a matching socket |
| | gent | | | |
| <i>deliver_in_udp_2</i> | udp: | network | nonur- | Get UDP datagram from host's in-queue but generate ICMP, as no matching socket |
| | gent | | | |
| <i>deliver_in_udp_3</i> | udp: | network | nonur- | Get UDP datagram from host's in-queue and drop as from a martian address |
| | gent | | | |

19.1.2 Rules

deliver_in_udp_1 **udp: network nonurgent** Get UDP datagram from host's in-queue and deliver it to a matching socket

$$h_0 \xrightarrow{\tau} h_0 \llbracket iq := iq';$$
$$socks := socks \oplus$$
$$\llbracket (sid, sock \ pr := \text{UDP_Sock}(rcvq')) \rrbracket$$
$$h_0 = h \llbracket iq := iq;$$
$$socks := socks \oplus$$
$$\llbracket (sid, sock \ pr := \text{UDP_Sock}(rcvq')) \rrbracket \wedge$$
$$rcvq' = rcvq @ [\text{DGRAM_MSG}(\llbracket data := data; is := \uparrow i_3; ps := ps_3 \rrbracket)] \wedge$$
$$\text{dequeue_iq}(iq, iq', \uparrow(\text{UDP}(\llbracket is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3; ps_2 := ps_4; data := data \rrbracket))) \wedge$$
$$(\exists (ifid, ifd) :: (h_0.ifds).i_4 \in ifd.ipset) \wedge$$
$$sid \in \text{lookup_udp } h_0.socks(i_3, ps_3, i_4, ps_4) h_0.bound h_0.arch \wedge$$
$$\mathbf{T} \wedge (* \text{ placeholder for "not a link-layer multicast or broadcast" } *)$$
$$\neg(\text{is_broadormulticast } h_0.ifds i_4) \wedge (* \text{ seems unlikely, since } i_1 \in \text{localLips } h.ifds *)$$
$$\neg(\text{is_broadormulticast } h_0.ifds i_3)$$

Description

At the head of the host's in-queue is a UDP datagram with source address ($\uparrow i_3, ps_3$), destination address ($\uparrow i_4, ps_4$), and data *data*. The destination IP address, i_4 , is an IP address for one of the host's interfaces and is not an IP- or link-layer broadcast or multicast address and neither is the source IP address, i_3 .

The UDP socket *sid* matches the address quad of the datagram (see [lookup_udp \(p86\)](#) for details). A τ transition is made. The datagram is removed from the host's in-queue, *iq*, and appended to the tail of the socket's receive queue, *rcvq'*, leaving the host with in-queue *iq'* and the socket with receive queue *rcvq'*.

deliver_in_udp_2 **udp: network nonurgent** Get UDP datagram from host's in-queue but generate ICMP, as no matching socket

$h \langle iq := iq \rangle \xrightarrow{\tau} h \langle iq := iq'; oq := \text{if } icmp_to_go \text{ then } oq' \text{ else } h.oq \rangle$

$dequeue_iq(iq, iq', \uparrow(\text{UDP}(\langle \langle is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3; ps_2 := ps_4; data := data \rangle \rangle)) \wedge$
 $lookup_udp \ h.socks(i_3, ps_3, i_4, ps_4)h.bound \ h.arch = \emptyset \wedge$
 $icmp = \text{ICMP}(\langle \langle is_1 := \uparrow i_4; is_2 := \uparrow i_3; is_3 := \uparrow i_3; is_4 := \uparrow i_4; ps_3 := ps_3; ps_4 := ps_4; proto := \text{PROTO_UDP}; seq := *; t := \text{ICMP_UNREACH}(\text{PORT}) \rangle \rangle) \wedge$
 $(enqueue_oq(h.oq, icmp, oq', \mathbf{T}) \vee icmp_to_go = \mathbf{F})$ (* non-deterministic ICMP generation *) ^
 $i_4 \in localLips \ h.ifds \wedge$
 $\mathbf{T} \wedge$ (* placeholder for "not a link-layer multicast or broadcast" *)
 $\neg(is_broadormulticast \ h.ifds \ i_4) \wedge$ (* seems unlikely, since $i_1 \in localLips \ h.ifds$ *)
 $\neg(is_broadormulticast \ h.ifds \ i_3)$

Description

At the head of the host's in-queue, iq , is a UDP datagram with source address $(\uparrow i_3, ps_3)$, destination address $(\uparrow i_4, ps_4)$, and data $data$. The destination IP address, i_4 , is an IP address for one of the host's interfaces and is neither a broadcast or multicast address; the source IP address, i_3 , is also not a broadcast or multicast address. None of the sockets in the host's finite map of sockets, $h.socks$, match the datagram (see `lookup_udp` (p86) for details).

A τ transition is made. The datagram is removed from the host's in-queue, leaving it with in-queue iq' . An ICMP Port-unreachable message may be generated and appended to the tail of the host's out-queue in response to the datagram.

deliver_in_udp_3 **udp: network nonurgent** Get UDP datagram from host's in-queue and drop as from a martian address

$h \langle iq := iq \rangle \xrightarrow{\tau} h \langle iq := iq' \rangle$

$dequeue_iq(iq, iq', \uparrow(\text{UDP } dgram)) \wedge$
 $dgram.is_2 = \uparrow i_2 \wedge$
 $is_1 = dgram.is_1 \wedge$
 $i_2 \in localLips(h.ifds) \wedge$
 $(\mathbf{F} \vee$
 $\neg(\mathbf{T} \wedge$
 $\neg(is_broadormulticast \ h.ifds \ i_2) \wedge$ (* seems unlikely, since $i_1 \in localLips \ h.ifds$ *)
 $\neg(is_1 = *) \wedge$
 $\neg is_broadormulticast \ h.ifds(\mathbf{the} \ is_1)$
 $)$
 $)$

Description

At the head of the host's in-queue, iq , is a UDP datagram with destination IP address $\uparrow i_2$ which is an IP address for one of the host's interfaces. Either i_2 is an IP-layer broadcast or multicast address, or the source IP address, is_1 , is not set or is an IP-layer broadcast or multicast address.

A τ transition is made. The datagram is dropped from the host's in-queue, leaving it with in-queue iq' .

Chapter 20

Host LTS: ICMP Input Processing

20.1 Input Processing (ICMP only)

20.1.1 Summary

| | | |
|--------------------------|-------------------------------|---|
| <i>deliver_in_icmp_1</i> | all: network nonurgent | Receive <i>ICMP_UNREACH_NET</i> etc for known socket |
| <i>deliver_in_icmp_2</i> | all: network nonurgent | Receive <i>ICMP_UNREACH_NEEDFRAG</i> for known socket |
| <i>deliver_in_icmp_3</i> | all: network nonurgent | Receive <i>ICMP_UNREACH_PORT</i> etc for known socket |
| <i>deliver_in_icmp_4</i> | all: network nonurgent | Receive <i>ICMP_PARAMPROB</i> etc for known socket |
| <i>deliver_in_icmp_5</i> | all: network nonurgent | Receive <i>ICMP_SOURCE_QUENCH</i> for known socket |
| <i>deliver_in_icmp_6</i> | all: network nonurgent | Receive and ignore other ICMP |
| <i>deliver_in_icmp_7</i> | all: network nonurgent | Receive and ignore invalid or unmatched ICMP |

20.1.2 Rules

deliver_in_icmp_1 **all: network nonurgent** Receive *ICMP_UNREACH_NET* etc for known socket

$$h_0 \xrightarrow{\tau} h \langle \langle socks := socks \oplus [(sid, sock')]; iq := iq'; oq := oq' \rangle \rangle$$
$$h_0 = h \langle \langle socks := socks \oplus [(sid, sock)]; iq := iq; oq := oq \rangle \rangle \wedge$$

dequeue_iq(iq, iq', ↑(ICMP icmp)) ∧
icmp.t ∈ {ICMP_UNREACH c |
c ∈ {NET; HOST; SRCFAIL; NET_UNKNOWN; HOST_UNKNOWN; ISOLATED;
TOSNET; TOSHOST; PREC_VIOLATION; PREC_CUTOFF}} ∧

$$icmp.is_3 = \uparrow i_3 \wedge$$
$$i_3 \notin \text{IN_MULTICAST} \wedge$$
$$sid \in \text{lookup_icmp } h_0.socks \text{ icmp } h_0.arch \ h_0.bound \wedge$$

(**case sock.pr of**
TCP_PROTO(tcp_sock) →
(∃icmp_seq.icmp.seq = ↑ icmp_seq ∧
if tcp_sock.cb.snd_una ≤ icmp_seq ∧ icmp_seq < tcp_sock.cb.snd_max **then**
if tcp_sock.st = ESTABLISHED **then**
sock' = sock ∧ (* ignore transient error while connected *)
oq' = oq
else if tcp_sock.st ∈ {CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED} ∧


```

    tcp_sock.cb.tt_rexmt ≠ * ∧ shift_of tcp_sock.cb.tt_rexmt > 3 ∧
    tcp_sock.cb.t_softerror ≠ * then
    tcp_drop_and_close h.arch(↑ EHOSTUNREACH)sock(sock', outsegs) ∧
    enqueue_and_ignore_fail h.arch h.rttab h.ifds outsegs oq oq'
else
    sock' = sock ⟨ pr := TCP_PROTO(tcp_sock
        ⟨ cb := tcp_sock.cb
          ⟨ t_softerror := ↑ EHOSTUNREACH⟩⟩⟩⟩ ∧
    oq' = oq
else
    (* Note the case where it is a syncache entry is not dealt with here: a syncache_unreach() should be
    done instead *)
    sock' = sock ∧
    oq' = oq ||
    UDP_PROTO(udp_sock) →
    if windows_arch h.arch then
        sock' = sock ⟨ pr := UDP_PROTO(udp_sock
            ⟨ rcvq := udp_sock.rcvq @ [(DGRAM_ERROR(⟨ e := ECONNRESET⟩))]⟩⟩⟩ ∧ oq' = oq
    else
        sock' = sock ⟨ es := ↑ ECONNREFUSED
            onlywhen((sock.is2 ≠ *) ∨ ¬(SO_BSDCOMPAT ∈ sock.sf.b))⟩ ∧ oq' = oq

```

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_UNREACH_NET.

deliver_in_icmp_2 **all: network nonurgent** Receive *ICMP_UNREACH_NEEDFRAG* for known socket

```

h0  $\xrightarrow{T}$  h ⟨ socks := socks ⊕
    [(sid, sock)];
    iq := iq';
    oq := oq'⟩

```

```

h0 = h ⟨ socks := socks ⊕
    [(sid, sock)];
    iq := iq;
    oq := oq⟩ ∧
dequeue_iq(iq, iq', ↑(ICMP icmp)) ∧
icmp.t = ICMP_UNREACH(NEEDEFRAG icmpmtu) ∧
(icmp.is3 = * ∨ the icmp.is3 ∉ IN_MULTICAST) ∧
sid ∈ lookup_icmp h0.socks icmp h0.arch h0.bound ∧
let nextmtu = if F ∧ (* Note this is a placeholder for "there is a host (not net) route for icmp.is4" *)
    F then (* Note this is a placeholder for "rmx.mtu not locked" *)
        let curmtu = 1492 in (* Note this value should be taken from rmx.mtu *)
        let nextmtu = case icmpmtu of
            ↑ mtu → w2n mtu
            || * → next_smaller(mtu_tab h0.arch)curmtu in
        if nextmtu < 296 then
            (* Note this should lock curmtu in rmxcache; and not change rmxcache MTU from
            curmtu *)
            ↑ curmtu
        else
            (* Note here, nextmtu should be stored in rmxcache *)
            ↑ nextmtu
    else
        * in
(case sock.pr of
    TCP_PROTO(tcp_sock) →

```

```

(∃icmp_seq.icmp.seq = ↑ icmp_seq ∧
if is_some icmp.is3 then
  (if tcp_sock.cb.snd_una ≤ icmp_seq ∧ icmp_seq < tcp_sock.cb.snd_max then
    if nextmtu = * then
      sock' = sock ⟨ pr := TCP_PROTO(tcp_sock
        ⟨ cb := tcp_sock.cb ⟨ t_maxseg := MSSDFLT⟩)⟩ ∧
      oq' = oq
    else
      let mss = min(sock.sf.n(SO_SNDBUF))
        (rounddown MCLBYTES
        (the nextmtu - 40 - (if tcp_sock.cb.tf_doing_tstamp then 12 else 0))) in
        (* BSD: TS, plus NOOP for alignment *)
      if mss ≤ tcp_sock.cb.t_maxseg then
        let sock'' = sock ⟨ pr := TCP_PROTO(tcp_sock
          ⟨ cb := tcp_sock.cb
            ⟨ t_maxseg := mss;
              t_rttseg := *;
              snd_next := tcp_sock.cb.snd_una
            ⟩⟩)⟩ in
          ∃sock''' outsegs tcp_sock'''.
          sock'''.pr = TCP_PROTO(tcp_sock''') ∧
          tcp_output_perhaps h.arch(ticks_of h.ticks)h.ifds sock''(sock''', outsegs) ∧
          enqueue_or_fail_sock(tcp_sock'''.st ∉ {CLOSED; LISTEN; SYN_SENT})
          h.arch h.rttab h.ifds outsegs oq
          sock'' sock'''(sock', oq')
        else
          sock' = sock ∧ oq' = oq
      else
        (* Note the case where it is a syncache entry is not dealt with here: a syncache_unreach() should be
        done instead *)
        sock' = sock ∧ oq' = oq)
    else
      sock' = sock ∧ oq' = oq) ||
  UDP_PROTO(udp_sock) →
if windows_arch h.arch then
  sock' = sock ⟨ pr := UDP_PROTO(udp_sock
    ⟨ rcvq := udp_sock.rcvq @ [(DGRAM_ERROR(⟨ e := EMSGSIZE⟩))]⟩)⟩ ∧ oq' = oq
else
  sock' = sock ⟨ es := ↑ EMSGSIZE⟩ ∧ oq' = oq)

```

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_MSGSIZE.

deliver_in_icmp_3 **all: network nonurgent** Receive ICMP_UNREACH_PORT etc for known socket

$$h_0 \xrightarrow{\tau} h \langle socks := socks \oplus [(sid, sock')]; iq := iq'; oq := oq' \rangle$$

$$h_0 = h \langle socks := socks \oplus [(sid, sock)]; iq := iq; oq := oq \rangle \wedge \text{dequeue_iq}(iq, iq', \uparrow(\text{ICMP } icmp)) \wedge icmp.t \in \{\text{ICMP_UNREACH } c \mid c \in \{\text{PROTOCOL; PORT; NET_PROHIB; HOST_PROHIB; FILTER_PROHIB}\} \wedge$$

```

icmp.is3 = ↑ i3 ∧
i3 ∉ IN_MULTICAST ∧
sid ∈ lookup_icmp h0.socks icmp h0.arch h0.bound ∧
(case sock.pr of
  TCP_PROTO(tcp_sock) →
    (∃ icmp_seq.icmp.seq = ↑ icmp_seq ∧
    if tcp_sock.cb.snd_una ≤ icmp_seq ∧ icmp_seq < tcp_sock.cb.snd_max then
      if tcp_sock.st = SYN_SENT then
        tcp_drop_and_close h.arch(↑ ECONNREFUSED) sock(sock', []) (* know from definition of
                                                                    tcp_drop_and_close that no
                                                                    segs will be emitted *)
      else
        sock' = sock ∧ oq' = oq
    else
      (* Note the case where it is a syncache entry is not dealt with here: a syncache_unreach() should be
      done instead *)
      sock' = sock ∧ oq' = oq) ||
  UDP_PROTO(udp_sock) →
    (if windows_arch h.arch then
      sock' = sock ⟨ pr := UDP_PROTO(udp_sock
      ⟨ rcvq := udp_sock.rcvq @ [(DGRAM_ERROR(⟨ e := ECONNRESET⟩))⟩] ⟩ ⟩ ∧
      oq' = oq
    else
      sock' = sock ⟨ es := ↑(ECONNREFUSED)
      onlywhen((sock.is2 ≠ *) ∨ ¬(SO_BSDCOMPAT ∈ sock.sf.b)) ⟩ ∧ oq' = oq))

```

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_UNREACH_PORT and PRC_UNREACH_ADMIN_PROHIB.

deliver_in_icmp_4 **all: network nonurgent** Receive ICMP_PARAMPROB etc for known socket

```

h0  $\xrightarrow{T}$  h ⟨ socks := socks ⊕
  [(sid, sock')];
  iq := iq';
  oq := oq' ⟩

```

```

h0 = h ⟨ socks := socks ⊕
  [(sid, sock)];
  iq := iq;
  oq := oq ⟩ ∧
dequeue_iq(iq, iq', ↑(ICMP icmp)) ∧
icmp.t ∈ {ICMP_PARAMPROB c |
  c ∈ {BADHDR; NEEDOPT}} ∧
icmp.is3 = ↑ i3 ∧
i3 ∉ IN_MULTICAST ∧
sid ∈ lookup_icmp h0.socks icmp h0.arch h0.bound ∧
(case sock.pr of

```

```

  TCP_PROTO(tcp_sock) →
    (∃ icmp_seq.icmp.seq = ↑ icmp_seq ∧
    if tcp_sock.cb.snd_una ≤ icmp_seq ∧ icmp_seq < tcp_sock.cb.snd_max then
      if tcp_sock.st ∈ {CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED} ∧
      tcp_sock.cb.tt_rexmt ≠ * ∧ shift_of tcp_sock.cb.tt_rexmt > 3 ∧
      tcp_sock.cb.t_softerror ≠ * then
        tcp_drop_and_close h.arch(↑ ENOPROTOOPT) sock(sock', outsegs) ∧
        enqueue_and_ignore_fail h.arch h.rttab h.ifds outsegs oq oq'
    else

```

```

sock' = sock ⟨ pr := TCP_PROTO(tcp_sock
                        ⟨ cb := tcp_sock.cb ⟨ t_softerror := ↑ ENOPROTOOPT⟩⟩)⟩ ∧
oq' = oq
else
sock' = sock ∧ oq' = oq ||
UDP_PROTO(udp_sock) →
  (if windows_arch h.arch then
    sock' = sock ⟨ pr := UDP_PROTO(udp_sock
                                    ⟨ rcvq := udp_sock.rcvq @ [(DGRAM_ERROR(⟨ e := ENOPROTOOPT⟩))])⟩)⟩ ∧
    oq' = oq
  else
    sock' = sock ⟨ es := ↑(ENOPROTOOPT)⟩ ∧ oq' = oq)

```

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_PARAMPROB.

deliver_in_icmp_5 **all: network nonurgent** Receive ICMP_SOURCE_QUENCH for known socket

```

h0  $\xrightarrow{T}$  h ⟨ socks := socks ⊕
                [(sid, sock')];
                iq := iq'⟩

```

```

h0 = h ⟨ socks := socks ⊕
          [(sid, sock)];
          iq := iq⟩ ∧
dequeue_iq(iq, iq', ↑(ICMP icmp)) ∧
icmp.t = ICMP_SOURCE_QUENCH QUENCH ∧
icmp.is3 = ↑ i3 ∧
i3 ∉ IN_MULTICAST ∧
sid ∈ lookup_icmp h0.socks icmp h0.arch h0.bound ∧
(case sock.pr of
  TCP_PROTO(tcp_sock) →
    (∃ icmpseq.icmp.seq = ↑ icmpseq ∧
    if tcp_sock.cb.snd_una ≤ icmpseq ∧ icmpseq < tcp_sock.cb.snd_max then
      sock' = sock ⟨ pr := TCP_PROTO(tcp_sock
                                      ⟨ cb := tcp_sock.cb
                                      ⟨ snd_cwnd := 1 * tcp_sock.cb.t_maxseg⟩)⟩)⟩
    (* Note the state of the TCP socket should be checked here. *)
    (* Note it might be necessary to make an allowance for local/remote connection? *)
  else
    (* Note the case where it is a syncache entry is not dealt with here: a syncache_unreach() should be
    done instead *)
    sock' = sock) ||
  UDP_PROTO(udp_sock) →
    (if windows_arch h.arch then
      sock' = sock ⟨ pr := UDP_PROTO(udp_sock
                                      ⟨ rcvq := udp_sock.rcvq @ [(DGRAM_ERROR(⟨ e := EHOSTUNREACH⟩))])⟩)⟩
    else
      sock' = sock ⟨ es := ↑(EHOSTUNREACH)⟩)

```

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_QUENCH.

deliver_in_icmp_6 **all: network nonurgent** Receive and ignore other ICMP

$h \langle iq := iq \rangle \xrightarrow{\tau} h \langle iq := iq' \rangle$

$\text{dequeue_iq}(iq, iq', \uparrow(\text{ICMP } icmp)) \wedge$
 $(icmp.t \in \{\text{ICMP_TIME_EXCEEDED INTRANS; ICMP_TIME_EXCEEDED REASS}\} \vee$
 $icmp.t \in \{\text{ICMP_UNREACH(OTHER } x) \mid x \in \text{UNIV}\} \vee$
 $icmp.t \in \{\text{ICMP_SOURCE_QUENCH(OTHER } x) \mid x \in \text{UNIV}\} \vee$
 $icmp.t \in \{\text{ICMP_TIME_EXCEEDED(OTHER } x) \mid x \in \text{UNIV}\} \vee$
 $icmp.t \in \{\text{ICMP_PARAMPROB(OTHER } x) \mid x \in \text{UNIV}\})$

Description If ICMP_TIME_EXCEEDED (either INTRANS or REASS), or if a bad code is received, then ignore silently.

deliver_in_icmp_7 **all: network nonurgent** Receive and ignore invalid or unmatched ICMP

$h \langle iq := iq \rangle \xrightarrow{\tau} h \langle iq := iq' \rangle$

$\text{dequeue_iq}(iq, iq', \uparrow(\text{ICMP } icmp)) \wedge$
 $(icmp.t \in \{\text{ICMP_UNREACH } c \mid \neg \exists x.c = \text{OTHER } x\} \vee$
 $icmp.t \in \{\text{ICMP_PARAMPROB } c \mid c \in \{\text{BADHDR; NEEDOPT}\}\} \vee$
 $icmp.t = \text{ICMP_SOURCE_QUENCH QUENCH}) \wedge$
(if $\exists icmpmtu.icmp.t = \text{ICMP_UNREACH(NEEDFRAG } icmpmtu)$ **then**
 $\exists i_3.icmp.is_3 = \uparrow i_3 \wedge i_3 \in \text{IN_MULTICAST}$
else
 $(icmp.is_3 = * \vee$
the $icmp.is_3 \in \text{IN_MULTICAST} \vee$
 $\neg(\exists(\text{sid}, s) :: (h.\text{socks}.$
 $s.is_1 = icmp.is_3 \wedge s.is_2 = icmp.is_4 \wedge$
 $s.ps_1 = icmp.ps_3 \wedge s.ps_2 = icmp.ps_4 \wedge$
 $\text{proto_of } s.pr = icmp.proto)))$

Description If the ICMP is a type we handle, but the source IP is IP 0 0 00 or a multicast address, or there's no matching socket, then drop silently. ICMP_UNREACH_NEEDFRAG is handled specially, since we do not care if it's IP 0 0 0 0, only if it's multicast.

Chapter 21

Host LTS: Network Input and Output

21.1 Input and Output (Network only)

21.1.1 Summary

| | | |
|------------------------|-------------------------------|------------------------------|
| <i>deliver_in_99</i> | all: network nonurgent | Really receive things |
| <i>deliver_in_99a</i> | all: network nonurgent | Ignore things not for us |
| <i>deliver_out_99</i> | all: network nonurgent | Really send things |
| <i>deliver_loop_99</i> | all: network nonurgent | Loop back a loopback message |

21.1.2 Rules

deliver_in_99 **all: network nonurgent** Really receive things

$h \langle iq := iq \rangle \xrightarrow{msg} h \langle iq := iq' \rangle$

sane_msg msg \wedge
 $\uparrow i_1 = msg.is_2 \wedge$
 $i_1 \in localLips(h.ifds) \wedge$
enqueue_iq(iq, msg, iq', queued)

Description Actually receive a message from the wire into the input queue. Note that if it cannot be queued (because the queue is full), it is silently dropped.

We only accept messages that are for this host. We also assert that any message we receive is well-formed (this excludes elements of type *msg* that have no physical realisation).

Note the delay in in-queuing the datagram is not modelled here.

deliver_in_99a **all: network nonurgent** Ignore things not for us

$h \langle iq := iq \rangle \xrightarrow{msg} h \langle iq := iq' \rangle$

$\uparrow i_1 = msg.is_2 \wedge$
 $i_1 \notin localLips(h.ifds) \wedge$
 $iq = iq'$

Description Do not accept messages that are not for this host.

deliver_out_99 **all: network nonurgent** Really send things

$$h \langle \langle oq := oq \rangle \rangle \xrightarrow{\overrightarrow{msg}} h \langle \langle oq := oq' \rangle \rangle$$

$$\text{dequeue_oq}(oq, oq', \uparrow msg) \wedge$$

$$(\exists i_2. msg.is_2 = \uparrow i_2 \wedge i_2 \notin \text{local_lips } h.ifds)$$

Description Actually emit a segment from the output queue.
 Note the delay in dequeuing the datagram is not modelled here.

deliver_loop_99 **all: network nonurgent** Loop back a loopback message

$$h \langle \langle iq := iq; \quad \xrightarrow{lbl} \quad h \langle \langle iq := iq';$$

$$oq := oq \rangle \rangle \quad \quad \quad oq := oq' \rangle \rangle$$

$$\text{dequeue_oq}(oq, oq', \uparrow msg) \wedge$$

$$(\exists i_2. msg.is_2 = \uparrow i_2 \wedge i_2 \in \text{local_lips } h.ifds) \wedge$$

$$(lbl = \text{if windows_arch } h.arch \text{ then } \tau$$

$$\quad \text{else } \overleftarrow{msg}) \wedge$$

$$\text{enqueue_iq}(iq, msg, iq', queued)$$

Description Deliver a loopback message (for loopback address, or any of our addresses) from the outqueue to the inqueue. (if we tagged each message in the outqueue with its interface, we'd just pick loopback-interface segments, but we do not, so we just discriminate on IP addresses).

Chapter 22

Host LTS: BSD Trace Records and Interface State Changes

22.1 Trace Records and Interface State Changes (BSD only)

22.1.1 Summary

| | | |
|--------------------|----------------------------|---|
| <i>trace_1</i> | all: misc nonurgent | Trace TCPCB state, ESTABLISHED or later |
| <i>trace_2</i> | all: misc nonurgent | Trace TCPCB state, pre-ESTABLISHED |
| <i>interface_1</i> | all: misc nonurgent | Change connectivity |

22.1.2 Rules

trace_1 **all: misc nonurgent** Trace TCPCB state, ESTABLISHED or later

$$h \xrightarrow{\text{LH_TRACE } tr} h$$

$sid \in \mathbf{dom}(h.socks) \wedge$
 $tr = (flav, sid, quad, st, cb) \wedge$
 $st \in \{\text{ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSING;}$
 $\quad \text{CLOSE_WAIT; LAST_ACK; TIME_WAIT}\} \wedge$
 $\text{tracesock_eq } tr \ sid(h.socks[sid])$

Description This rule exposes certain of the fields of the socket and TCPCB, to allow open-box testing. Note that although the label carries an entire TCPCB, only certain selected fields are constrained to be equal to the actual TCPCB. See [tracesock_eq \(p63\)](#) and [tracecb_eq \(p62\)](#) for details.

Checking trace equality is problematic as BSD generates trace records that fall logically inbetween the atomic transitions in this model. This happens frequently when in a state before ESTABLISHED. We only check for equality when we are in ESTABLISHED or later states.

trace_2 **all: misc nonurgent** Trace TCPCB state, pre-ESTABLISHED

$$h \xrightarrow{\text{LH_TRACE } tr} h$$

$sid \in \mathbf{dom}(h.socks) \wedge$
 $tr = (flav, sid, quad, st, cb) \wedge$
 $st \notin \{\text{ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSING;}$
 $\quad \text{CLOSE_WAIT; LAST_ACK; TIME_WAIT}\} \wedge$


```

(st = CLOSED ∨ (* BSD emits one of these each time a tcpcb is created, eg at end of 3WHS *))
((∃ sock tcp_sock.
  sock = (h.socks[sid]) ∧
  proto_of sock.pr = PROTO_TCP ∧
  tcp_sock = tcp_sock_of sock ∧
  (case quad of
    ↑(is1, ps1, is2, ps2) → if flav = TA_DROP ∨ tcp_sock.st = CLOSED then T
    else
      is1 = sock.is1 ∧ ps1 = sock.ps1 ∧ is2 = sock.is2 ∧ ps2 = sock.ps2 ||
      * → T) ∧
  (st = tcp_sock.st ∨ tcp_sock.st = CLOSED))))

```

interface_1 **all: misc nonurgent** Change connectivity

$h \langle ifds := ifds \rangle \xrightarrow{\text{LH_INTERFACE}(ifid, up)} h \langle ifds := ifds' \rangle$

$ifid \in \mathbf{dom}(ifds) \wedge$
 $ifds' = ifds \oplus (ifid, (ifds[ifid]) \langle up := up \rangle)$

Description Allow interfaces to be externally brought up or taken down.

Chapter 23

Host LTS: Time Passage

23.1 Time Passage auxiliaries (TCP and UDP)

Time passage is a *function*, completely deterministic. Any nondeterminism must occur as a result of a tau (or other) transition.

In the present semantics, time passage merely:

1. decrements all timers uniformly
2. prevents time passage if a timer reaches zero
3. prevents time passage if an urgent action is enabled.

We model the first two points with functions *Time_Pass_**, for various types *. These functions return an option type: if the result is NONE then time may not pass for the given duration. Essentially they pick out everything in a host state of type '*a* timed, and do something with it.

We treat the last point in the rule *epsilon_1* (p348) itself, below.

23.1.1 Summary

| | |
|------------------------------|---|
| <i>Time_Pass_timedoption</i> | time passes for an ' <i>a</i> timed option value |
| <i>Time_Pass_tcpcb</i> | time passes for a tcp control block |
| <i>Time_Pass_socket</i> | time passes for a socket |
| <i>fmap_every</i> | apply <i>f</i> to range of finite map, and succeed if each application succeeds |
| <i>fmap_every_pred</i> | apply <i>f</i> to range of finite map, and succeed if each application succeeds |
| <i>Time_Pass_host</i> | time passes for a host |

23.1.2 Rules

– **time passes for an '*a* timed option value :**
(*Time_Pass_timedoption* : duration → '*a* timed option → '*a* timed option option)
dur x0
= **case** *x0* **of**
 * → ↑ * ||
 ↑ *x* → (**case** *Time_Pass_timed dur x of*
 * → * ||
 ↑ *x0'* → ↑(↑ *x0'*))

– **time passes for a tcp control block :**

```

(Time_Pass_tcpcb : duration → tcpcb → tcpcb set option)(* recall: 'a set == 'a -> bool *)
dur cb
= let tt_rexmt' = Time_Pass_timedoption dur cb.tt_rexmt
and tt_keep' = Time_Pass_timedoption dur cb.tt_keep
and tt_2msl' = Time_Pass_timedoption dur cb.tt_2msl
and tt_delack' = Time_Pass_timedoption dur cb.tt_delack
and tt_conn_est' = Time_Pass_timedoption dur cb.tt_conn_est
and tt_fin_wait_2' = Time_Pass_timedoption dur cb.tt_fin_wait_2
and ts_recent's = Time_Pass_timewindow dur cb.ts_recent
and t_badrxtwin's = Time_Pass_timewindow dur cb.t_badrxtwin
and t_idletime's = Time_Pass_stopwatch dur cb.t_idletime
in
if is_some tt_rexmt' ∧
  is_some tt_keep' ∧
  is_some tt_2msl' ∧
  is_some tt_delack' ∧
  is_some tt_conn_est' ∧
  is_some tt_fin_wait_2'
then
  ↑(λcb'.
    choose ts_recent' :: ts_recent's.
    choose t_badrxtwin' :: t_badrxtwin's.
    choose t_idletime' :: t_idletime's.
    cb' =
    cb ( (* not going to list everything here; too much! *)
      tt_rexmt := the tt_rexmt';
      tt_keep := the tt_keep';
      tt_2msl := the tt_2msl';
      tt_delack := the tt_delack';
      tt_conn_est := the tt_conn_est';
      tt_fin_wait_2 := the tt_fin_wait_2';
      ts_recent := ts_recent';
      t_badrxtwin := t_badrxtwin';
      t_idletime := t_idletime'
    ))
else
  *

```

– **time passes for a socket :**

```

(Time_Pass_socket : duration → socket → socket set option)
dur s
= case s.pr of UDP_PROTO(udp) → ↑{s}
|| TCP_PROTO(tcp_s) →
  let cb's = Time_Pass_tcpcb dur tcp_s.cb
  in
  if is_some cb's
  then
    ↑(λs'.
      choose cb' :: the cb's.
      s' =
      s ( (* fid unchanged *)
        (* sf unchanged *)
        (* is1,ps1,is2,ps2 unchanged *)
        (* es unchanged *)
        pr := TCP_PROTO(tcp_s ( cb := cb' ))
      ))

```

else

*

– apply f to range of finite map, and succeed if each application succeeds :

(fmap_every : ('a → 'b option) → ('c ↦ 'a) → ('c ↦ 'b) option)

f $fm =$

let $fm' = f$ o_ f fm

in

if $*$ ∈ rng(fm')

then *

else ↑(the o_ f fm')

– apply f to range of finite map, and succeed if each application succeeds :

(fmap_every_pred : ('a → 'b set option) → ('c ↦ 'a) → ('c ↦ 'b)set option)

f $fm =$

if $\exists y.y \in \text{rng}(fm) \wedge f y = *$ then

*

else

↑{ $fm' \mid \text{dom}(fm) = \text{dom}(fm') \wedge$
 $\forall x.x \in \text{dom}(fm) \implies fm'[x] \in (\text{the}(f(fm[x])))$ }

– time passes for a host :

(Time_Pass_host : duration → host → host set option)

dur h

= let $ts' = \text{fmap_every}(\text{Time_Pass_timed } dur)h.ts$

and $socks's = \text{fmap_every_pred}(\text{Time_Pass_socket } dur)h.socks$

and $iq' = \text{Time_Pass_timed } dur$ $h.iq$

and $oq' = \text{Time_Pass_timed } dur$ $h.oq$

and $ticks's = \text{Time_Pass_ticker } dur$ $h.ticks$

in

if is_some $ts' \wedge$

is_some $socks's \wedge$

is_some $iq' \wedge$

is_some oq'

then

↑($\lambda h'.$

choose $socks' :: \text{the } socks's.$

choose $ticks' :: ticks's.$

$h' =$

h \ll (* arch unchanged *)

(* ifds unchanged *)

$ts := \text{the } ts';$

(* files unchanged *)

$socks := socks';$

(* listen unchanged *)

(* bound unchanged *)

$iq := \text{the } iq';$

$oq := \text{the } oq';$

$ticks := ticks'$

(* fds unchanged *)

```

    else
      *)

```

23.2 Host transitions with time (TCP and UDP)

We now build the relation \Rightarrow , which includes time transitions, from the relation \rightarrow , which is instantaneous. This avoids circularity (or at best inductiveness) in the definition of the transition relation.

23.2.1 Summary

| | | |
|------------------|----------------------------|----------------------------------|
| <i>epsilon_1</i> | all: misc nonurgent | Time passes |
| <i>epsilon_2</i> | all: misc nonurgent | Inductively defined time passage |
| <i>rn</i> | rp: rc | |

23.2.2 Rules

```

epsilon_1 all: misc nonurgent Time passes
h  $\xrightarrow{dur}$  h'

let hs' = Time_Pass_host dur h in
  is_some hs'  $\wedge$ 
  h'  $\in$  (the hs')  $\wedge$ 
   $\neg(\exists rn\ rp\ rc\ lbl\ h'.rn / * rp, rc * / h \xrightarrow{lbl} h' \wedge \text{is\_urgent } rc)$ 

```

Description Allow time to pass for *dur* seconds. This is only enabled if the host state is not urgent, i.e. if no urgent rule can fire. Notice that, apart from when a timer becomes zero, a host state never becomes urgent due merely to time passage. This means we need only test for urgency at the beginning of the time interval, not throughout it.

```

epsilon_2 all: misc nonurgent Inductively defined time passage
h  $\xrightarrow{dur}$  h'

( $\exists h_1\ h_2\ dur'\ dur''.$ 
  dur' < dur  $\wedge$ 
  ( $\exists rn\ rp\ rc.rn / * rp, rc * / h \xrightarrow{dur'} h_1$ )  $\wedge$ 
  ( $\exists rn\ rp\ rc.rn / * rp, rc * / h_1 \xrightarrow{\tau} h_2$ )  $\wedge$ 
  dur' + dur'' = dur  $\wedge$ 
  ( $\exists rn\ rp\ rc.rn / * rp, rc * / h_2 \xrightarrow{dur''} h'$ )
)

```

Description Combine time passage and τ transitions.

$$rn \quad \underline{rp: rc}$$

$$h \xrightarrow{\underline{lbl}} h'$$

$$rn / * rp, rc * /$$

$$h$$

$$\xrightarrow{\underline{lbl}}$$

$$h'$$

Description Embed all non-time transitions in the full LTS

Part XIV

TCP1_evalSupport

Chapter 24

Initial state

This file defines a function to construct certain initial host states for use in automated trace checking, along with other constants used in typical traces. The interfaces, routing table and some host fields are taken from the `initial_host` line at the start of a valid trace.

24.1 Initial state (TCP and UDP)

The initial state of a host.

24.1.1 Summary

| | |
|-----------------------|--|
| <i>simple_ifd_eth</i> | simple ethernet interface |
| <i>simple_ifd_lo</i> | simple loopback interface |
| <i>simple_rttab</i> | simple routing table |
| <i>tid_initial</i> | initial thread id |
| <i>simple_host</i> | simple host state |
| <i>dummy_cb</i> | |
| <i>dummy_socket</i> | minimal socket |
| <i>dummy_sockets</i> | |
| <i>initial_host</i> | function to construct an initial host for trace checking |

24.1.2 Rules

```
– simple ethernet interface :  
simple_ifd_eth i = (ETH 0, { ipset := {i}; primary := i; netmask := NETMASK 24; up := T})
```

```
– simple loopback interface :  
simple_ifd_lo = (LO, { ipset := LOOPBACK_ADDRS; primary := ip_localhost;  
                  netmask := NETMASK 8; up := T})
```

```
– simple routing table :  
simple_rttab = [ { destination_ip := ip_localhost;  
                  destination_netmask := NETMASK 8;  
                  ifid := LO };  
          { destination_ip := IP 0 0 0 0;
```



```

    destination_netmask := NETMASK 0;
    ifid := ETH 0}]

```

```

- initial thread id :

```

```

tid_initial = TID 0

```

```

- simple host state :

```

```

simple_host i tick0 remdr0 =
  ⟨ arch := FREEBSD_4_6_RELEASE;
    privs := F;
    ifds := ∅ ⊕ [simple_ifd_lo; simple_ifd_eth i];
    rttab := simple_rttab;
    ts := ∅ ⊕ (tid_initial ↦ (RUN)never_timer);
    files := ∅;
    socks := ∅;
    listen := [];
    bound := [];
    iq := ([])never_timer;
    oq := ([])never_timer;
    bndlm := bandlim_state_init;
    ticks := TICKER(tick0, remdr0, tickintvlmin, tickintvlmax);
    fds := ∅⟩

```

```

- :

```

```

dummy_cb = ⟨ tt_rexmt := *;
             tt_2msl := *;
             tt_conn_est := *;
             tt_delack := *;
             tt_keep := *;
             tt_fin_wait_2 := *;
             t_idletime := STOPWATCH(0, 1, 1);
             t_badrxtwin := TIMEWINDOWCLOSED;
             ts_recent := TIMEWINDOWCLOSED⟩

```

```

- minimal socket :

```

```

dummy_socket(is, p) =
  ⟨ fid := *;
    sf := ⟨ b := λx.F; n := λx.0; t := λx.∞⟩;
    is1 := is;
    ps1 := ↑ p;
    is2 := *;
    ps2 := *;
    pr := TCP_PROTO(⟨ st := LISTEN;
                      cb := dummy_cb;
                      lis := ↑ ⟨ q0 := []; q := []; qlimit := 10⟩
                    ⟩)
  ⟩

```

Description This is a pretty minimally-defined socket, just enough to say "this port is bound".

– :

```
dummy_sockets n[] = [] ∧
dummy_sockets n(p :: ps) = (SID n, dummy_socket p) :: dummy_sockets(n + 1)ps
```

– **function to construct an initial host for trace checking :**

```
initial_host(i : ip)(t : tid)(arch : arch)(ispriv : bool)
  (heldports : (ip option#port)list)(ifaces : (ifid#ifd)list)
  (rt : routing_table)
  (init_tick : ts_seq)
  (init_tick_remdr : duration)
= simple_host i init_tick init_tick_remdr ⟨
  arch := arch;
  privs := ispriv;
  ifds := ∅ ⊕ ifaces;
  rttab := rt;
  ts := ∅ ⊕ (t ↦ (RUN)never_timer);
  fds := case arch of
  (* per architecture, note down FDs preallocated for internal use by
  OCaml or the test harness *)
  LINUX_2_4_20_8 →
  ∅ ⊕ [(FD 0, FID 0);
  (FD 1, FID 0);
  (FD 2, FID 0);
  (FD 3, FID 0);
  (FD 4, FID 0);
  (FD 5, FID 0);
  (FD 6, FID 0);
  (FD 1000, FID 0)
  ]
  || FREEBSD_4_6_RELEASE →
  ∅ ⊕ [(FD 0, FID 0);
  (FD 1, FID 0);
  (FD 2, FID 0);
  (FD 3, FID 0);
  (FD 4, FID 0);
  (FD 5, FID 0);
  (FD 6, FID 0);
  (FD 7, FID 0)
  ]
  || WINXP_PROF_SP1 →
  ∅; (* Windows FDs are not allocated in order, so there's no need to
  specify anything here. *)
  files := ∅ ⊕ (FID 0,
  FILE(FT_CONSOLE, ⟨ b := λx.F ⟩));
  socks := ∅ ⊕ (dummy_sockets 0 heldports)
  ⟩
```

Index

- abstime*, 20
- accept_1*, 126
- accept_2*, 127
- accept_3*, 127
- accept_4*, 128
- accept_5*, 129
- accept_6*, 129
- accept_7*, 130
- accept_incoming_q*, 91
- accept_incoming_q0*, 91
- andThen*, 104
- arch*, 60
- assert*, 104
- assert_failure*, 104
- ASSERTION_FAILURE*, 4
- auto_outroute*, 82
- autobind*, 85

- backlog_fudge*, 75
- badf_1*, 274
- bandlim_reason*, 61
- bandlim_rst_ok*, 95
- bandlim_rst_ok_always*, 94
- bandlim_rst_ok_simple*, 94
- bandlim_state_init*, 94
- bind_1*, 133
- bind_2*, 134
- bind_3*, 134
- bind_5*, 135
- bind_7*, 135
- bind_9*, 135
- bound_after*, 85
- bound_port_allowed*, 85
- bound_ports_protocol_autobind*, 85
- bsd_arch*, 79
- bsd_make_phantom_segment*, 109
- BSD_RTTVAR_BUG*, 66

- calculate_bsd_rcv_wnd*, 93
- calculate_buf_sizes*, 93
- calculate_tcp_options_len*, 92
- chooseM*, 104
- clip_int_to_num*, 2
- close_1*, 138
- close_10*, 144
- close_2*, 138
- close_3*, 139
- close_4*, 140
- close_5*, 141
- close_6*, 142
- close_7*, 142
- close_8*, 143
- computed_rto*, 97
- computed_rxtcur*, 97
- CONCAT_OPTIONAL*, 3
- connect_1*, 148
- connect_10*, 161
- connect_2*, 152
- connect_3*, 152
- connect_4*, 153
- connect_4a*, 154
- connect_5*, 154
- connect_5a*, 155
- connect_5b*, 156
- connect_5c*, 157
- connect_5d*, 157
- connect_6*, 158
- connect_7*, 158
- connect_8*, 159
- connect_9*, 160
- cont*, 104

- decr_list*, 3
- deliver_in_1*, 279
- deliver_in_1b*, 283
- deliver_in_2*, 285
- deliver_in_2a*, 290
- deliver_in_3*, 291
- deliver_in_3a*, 309
- deliver_in_3b*, 310
- deliver_in_3c*, 311
- deliver_in_4*, 312
- deliver_in_5*, 313
- deliver_in_6*, 313
- deliver_in_7*, 314
- deliver_in_7a*, 315
- deliver_in_7b*, 316
- deliver_in_7c*, 317
- deliver_in_7d*, 318
- deliver_in_8*, 319
- deliver_in_9*, 320
- deliver_in_99*, 341
- deliver_in_99a*, 341
- deliver_in_icmp_1*, 335
- deliver_in_icmp_2*, 336
- deliver_in_icmp_3*, 337
- deliver_in_icmp_4*, 338
- deliver_in_icmp_5*, 339
- deliver_in_icmp_6*, 339
- deliver_in_icmp_7*, 340

- deliver_in_udp_1*, 333
- deliver_in_udp_2*, 333
- deliver_in_udp_3*, 334
- deliver_loop_99*, 342
- deliver_out_1*, 323
- deliver_out_99*, 341
- dequeue*, 90
- dequeue_iq*, 90
- dequeue_oq*, 90
- dgram*, 58
- dgram_error*, 58
- dgram_msg*, 58
- di3_ackstuff*, 298
- di3_datastuff*, 304
- di3_datastuff_really*, 300
- di3_newackstuff*, 295
- di3_socks_update*, 308
- di3_ststuff*, 305
- di3_topstuff*, 294
- diqmax*, 67
- disconnect_1*, 164
- disconnect_2*, 165
- disconnect_3*, 166
- disconnect_4*, 163
- disconnect_5*, 164
- do_tcp_options*, 92
- doqmax*, 67
- dosend*, 96
- DROP*, 3
- drop_from_q0*, 91
- dropwithreset*, 120
- dropwithreset_ignore_fail*, 120
- dschedmax*, 67
- dtsinval*, 73
- dummy_cb*, 352
- dummy_socket*, 352
- dummy_sockets*, 353
- dup_1*, 167
- dup_2*, 167
- dupfd_1*, 169
- dupfd_3*, 170
- dupfd_4*, 170
- duration*, 20

- emit_segs*, 105
- emit_segs_pred*, 105
- enqueue*, 90
- enqueue_and_ignore_fail*, 118
- enqueue_each_and_ignore_fail*, 118
- enqueue_iq*, 90
- enqueue_list*, 91
- enqueue_list_qinfo*, 91
- enqueue_oq*, 90
- enqueue_oq_bndlim_rst*, 95
- enqueue_oq_list*, 91
- enqueue_oq_list_qinfo*, 91
- enqueue_or_fail*, 118
- enqueue_or_fail_sock*, 118
- ephemeral_ports*, 69

- epsilon_1*, 348
- epsilon_2*, 348
- err*, 16
- error*, 7
- expand_cwnd*, 99

- fast_timer*, 88
- FAST_TIMER_INTVL*, 68
- FAST_TIMER_MODEL_INTVL*, 68
- fd*, 14
- fd_op*, 35
- FD_SETSIZE*, 69
- fd_sockop*, 35
- fdle*, 83
- fdlt*, 83
- ff_default*, 71
- ff_default_b*, 71
- fid*, 53
- fid_ref_count*, 84
- File*, 53
- file*, 53
- filebflag*, 14
- fileflags*, 53
- filetype*, 53
- fm_exists*, 2
- fmap_every*, 347
- fmap_every_pred*, 347
- funupd*, 2
- funupd_list*, 2
- fuzzy_timer*, 47

- get_cb*, 104
- get_sock*, 104
- get_tcp_sock*, 104
- getfileflags_1*, 171
- getifaddrs_1*, 173
- getpeername_1*, 175
- getpeername_2*, 176
- getsockbopt_1*, 178
- getsockbopt_2*, 178
- getsockerr_1*, 180
- getsockerr_2*, 180
- getsocklistening_1*, 182
- getsocklistening_2*, 183
- getsocklistening_3*, 182
- getsockname_1*, 185
- getsockname_2*, 185
- getsockname_3*, 186
- getsocknopt_1*, 188
- getsocknopt_4*, 188
- getsocktopt_1*, 190
- getsocktopt_4*, 191

- host*, 61
- hostThreadState*, 61
- HZ*, 68

- icmp_paramprob_code*, 30
- icmp_redirect_code*, 29

- icmp_source_quench_code*, 29
- icmp_time_exceeded_code*, 30
- icmp_unreach_code*, 29
- icmpDatagram*, 30
- icmpType*, 30
- if_any*, 80
- if_broadcast*, 80
- ifd*, 60
- ifid*, 13
- ifid_up*, 82
- in_local*, 80
- in_loopback*, 80
- IN_MULTICAST*, 80
- INADDR_BROADCAST*, 80
- INFINITE_RESOURCES*, 66
- initial_cb*, 101
- initial_host*, 353
- inqueue_timer*, 88
- INSERT_ORDERED*, 3
- interface_1*, 344
- intr_1*, 275
- iobc*, 57
- IP*, 80
- ip*, 13
- ip_localhost*, 80
- is_broadormulticast*, 81
- is_localnet*, 80
- is_urgent*, 39

- kern_timer*, 88
- KERN_TIMER_INTVL*, 68
- KERN_TIMER_MODEL_INTVL*, 68

- leastfd*, 83
- left_shift_num*, 2
- Lhost0*, 38
- LIB_interface*, 33
- linux_arch*, 79
- listen_1*, 193
- listen_1b*, 194
- listen_1c*, 194
- listen_2*, 195
- listen_3*, 195
- listen_4*, 196
- listen_5*, 197
- listen_7*, 197
- local_ips*, 80
- local_primary_ips*, 80
- lookup_icmp*, 87
- lookup_udp*, 86
- LOOPBACK_ADDRS*, 80
- loopback_on_wire*, 83

- make_ack_segment*, 108
- make_rst_segment_from_cb*, 109
- make_rst_segment_from_seg*, 110
- make_syn_ack_segment*, 107
- make_syn_segment*, 106
- MAP_OPTIONAL*, 3

- mask*, 80
- mask_bits*, 80
- match_score*, 85
- MCLBYTES*, 70
- mlift_dropafterack_or_fail*, 120
- mlift_tcp_output_perhaps_or_fail*, 118
- mliftc*, 105
- mliftc_bndlm*, 105
- mode_of*, 97
- modify_cb*, 104
- modify_sock*, 104
- modify_tcp_sock*, 104
- msg*, 31
- msg_is1*, 31
- msg_is2*, 31
- msgbflag*, 15
- MSIZE*, 70
- MSSDFLT*, 74
- mtu_tab*, 99

- netmask*, 14
- never_timer*, 47
- next_smaller*, 99
- nextfd*, 83
- nonurgent*, 39
- NOTIN'*, 3
- notsock_1*, 275
- num_floor*, 2
- num_floor_and_frac*, 2

- onlywhen*, 2
- oob_extra_sndbuf*, 70
- OPEN_MAX*, 69
- OPEN_MAX_FD*, 69
- opttorel*, 46
- ORDERINGS*, 3
- outqueue_timer*, 88
- outroute*, 82
- outroute_ifids*, 81

- port*, 13
- privileged_ports*, 69
- proto_eq*, 59
- proto_of*, 59
- protocol*, 29
- protocol_info*, 58
- pselect_1*, 200
- pselect_2*, 203
- pselect_3*, 203
- pselect_4*, 204
- pselect_5*, 205
- pselect_6*, 205
- pselect_timeo_t_max*, 73

- real_mult_time*, 19
- real_of_int*, 2
- realopt_of_time*, 20
- recv_1*, 209
- recv_11*, 221

- recv_12*, 222
- recv_13*, 222
- recv_14*, 223
- recv_15*, 224
- recv_16*, 224
- recv_17*, 225
- recv_2*, 211
- recv_20*, 225
- recv_21*, 227
- recv_22*, 227
- recv_23*, 228
- recv_24*, 228
- recv_3*, 211
- recv_4*, 213
- recv_5*, 214
- recv_6*, 214
- recv_7*, 215
- recv_8*, 215
- recv_8a*, 216
- recv_9*, 217
- REPLICATE*, 3
- resourcefail_1*, 276
- resourcefail_2*, 276
- retType*, 34
- return_1*, 274
- rxmtmode*, 55
- right_shift_num*, 2
- rn*, 348
- rollback_tcp_output*, 117
- rounddown*, 2
- roundup*, 2
- route_and_enqueue_oq*, 91
- routeable*, 81
- routing_table_entry*, 60
- rttinf*, 55
- rule_cat*, 39
- rule_ids*, 42
- rule_proto*, 39
- rule_status*, 39

- sane_msg*, 31
- sane_seg*, 27
- sane_socket*, 84
- sane_udpdgm*, 27
- SB_MAX*, 70
- sched_timer*, 88
- send_1*, 231
- send_10*, 244
- send_11*, 245
- send_12*, 246
- send_13*, 247
- send_14*, 247
- send_15*, 248
- send_16*, 249
- send_17*, 249
- send_18*, 250
- send_19*, 250
- send_2*, 234
- send_21*, 251
- send_22*, 252
- send_23*, 253
- send_3*, 235
- send_3a*, 235
- send_4*, 236
- send_5*, 237
- send_5a*, 237
- send_6*, 237
- send_7*, 238
- send_8*, 239
- send_9*, 243
- send_queue_space*, 93
- seq32*, 21
- seq32_coerce*, 21
- seq32_diff*, 21
- seq32_fromto*, 21
- seq32_geq*, 21
- seq32_gt*, 21
- seq32_leq*, 21
- seq32_lt*, 21
- seq32_max*, 21
- seq32_min*, 21
- seq32_minus*, 21
- seq32_minus'*, 21
- seq32_plus*, 21
- seq32_plus'*, 21
- setfileflags_1*, 254
- setsockbopt_1*, 256
- setsockbopt_2*, 257
- setsocknopt_1*, 259
- setsocknopt_2*, 259
- setsocknopt_4*, 260
- setsocktopt_1*, 262
- setsocktopt_4*, 262
- setsocktopt_5*, 263
- sf_default*, 72
- sf_default_b*, 71
- sf_default_n*, 71
- sf_default_t*, 72
- sf_max_n*, 72
- sf_min_n*, 72
- sharp_timer*, 47
- shift_of*, 97
- shutdown_1*, 265
- shutdown_2*, 266
- shutdown_3*, 266
- shutdown_4*, 267
- sid*, 53
- signal*, 10
- simple_host*, 352
- simple_ifd_eth*, 351
- simple_ifd_lo*, 351
- simple_limit*, 94
- simple_rttab*, 351
- slow_timer*, 88
- SLOW_TIMER_INTVL*, 68
- SLOW_TIMER_MODEL_INTVL*, 68
- sndrcv_timeo_t_max*, 73

- Sock*, 59
- socketmark_1*, 269
- socketmark_2*, 269
- sockbflag*, 14
- socket*, 59
- socket_1*, 272
- socket_2*, 273
- socket_listen*, 57
- sockflags*, 58
- socknflag*, 15
- socktflag*, 15
- socktype*, 16
- soexceptional*, 203
- SOMAXCONN*, 70
- soreadable*, 202
- sowriteable*, 202
- SPLIT*, 3
- SPLIT_REV*, 3
- SPLIT_REV_0*, 3
- SS_FLTSZ*, 74
- SS_FLTSZ_LOCAL*, 74
- start_tt_persist*, 97
- start_tt_rexmt*, 97
- start_tt_rexmt_gen*, 97
- start_tt_rexmtsyzn*, 97
- stop*, 104
- stopwatch*, 51
- stopwatch_val_of*, 51
- stopwatch_zero*, 68
- stopwatchfuzz*, 68

- TAKE*, 3
- TAKEWHILE*, 3
- TAKEWHILE_REV*, 3
- tcp_backoffs*, 96
- TCP_BSD_BACKOFFS*, 76
- tcp_close*, 121
- TCP_DO_NEWRENO*, 74
- tcp_drop_and_close*, 121
- TCP_LINUX_BACKOFFS*, 76
- TCP_MAXRXTSHIFT*, 77
- TCP_MAXWIN*, 73
- TCP_MAXWINSYSCALE*, 73
- tcp_output_perhaps*, 116
- tcp_output_really*, 113
- tcp_output_required*, 111
- TCP_Q0MAXLIMIT*, 74
- TCP_Q0MINLIMIT*, 74
- tcp_reass*, 100
- tcp_reass_prune*, 101
- tcp_seq_foreign*, 22
- tcp_seq_foreign_to_local*, 22
- tcp_seq_local*, 22
- tcp_seq_local_to_foreign*, 22
- TCP_Sock*, 59
- TCP_Sock0*, 59
- tcp_sock_of*, 59
- tcp_socket*, 58
- tcp_socket_best_match*, 86
- tcp_syn_backoffs*, 96
- TCP_SYN_BSD_BACKOFFS*, 77
- TCP_SYN_LINUX_BACKOFFS*, 77
- TCP_SYN_WINXP_BACKOFFS*, 77
- TCP_SYNACKMAXRXTSHIFT*, 77
- TCP_WINXP_BACKOFFS*, 76
- tcpcb*, 55
- tcpForeign*, 22
- tcpLocal*, 22
- tcpReassSegment*, 54
- tcpSegment*, 26
- tcpstate*, 54
- TCPTV_DELACK*, 75
- TCPTV_KEEP_IDLE*, 76
- TCPTV_KEEP_INIT*, 76
- TCPTV_KEEPCNT*, 76
- TCPTV_KEEPINTVL*, 76
- TCPTV_MAXIDLE*, 76
- TCPTV_MIN*, 75
- TCPTV_MSL*, 76
- TCPTV_PERSMAX*, 76
- TCPTV_PERSMIN*, 76
- TCPTV_REXMTMAX*, 75
- TCPTV_RTOBASE*, 75
- TCPTV_RTTVARBASE*, 75
- test_outroute*, 82
- test_outroute_ip*, 82
- the_time*, 20
- tick_imax*, 50
- tick_imin*, 50
- ticker*, 50
- ticker_ok*, 50
- tickintulmax*, 68
- tickintulmin*, 68
- ticks_of*, 50
- tid*, 16
- tid_initial*, 352
- time*, 19
- time_gt*, 19
- time_gte*, 19
- time_lt*, 19
- time_lte*, 19
- time_max*, 19
- time_min*, 19
- time_minus_dur*, 19
- time_of_tptime*, 89
- time_of_tptimeopt*, 89
- time_pass_additive*, 45
- Time_Pass_host*, 347
- Time_Pass_socket*, 346
- Time_Pass_stopwatch*, 51
- Time_Pass_tcpcb*, 345
- Time_Pass_ticker*, 50
- Time_Pass_timed*, 48
- Time_Pass_timedoption*, 345
- Time_Pass_timer*, 47
- Time_Pass_timewindow*, 49
- time_pass_trajectory*, 46

time_plus_dur, 19
time_zero, 20
timed, 48
timed_expires, 48
timed_timer_of, 48
timed_val_of, 48
timer, 47
timer_expires, 47
timer_tt_2msl_1, 330
timer_tt_conn_est_1, 331
timer_tt_delack_1, 331
timer_tt_fin_wait_2_1, 331
timer_tt_keep_1, 329
timer_tt_persist_1, 329
timer_tt_rexmt_1, 327
timer_tt_rexmtsyn_1, 325
timewindow, 49
timewindow_open, 49
timewindow_val_of, 49
TLang, 17
TLang_type, 16
tlang_typing, 17
tltimeopt_of_time, 89
tltimeopt_wf, 89
trace_1, 343
trace_2, 343
tracecb_eq, 62
traceflavour, 62
tracesock_eq, 63
ts_seq, 23
tstamp, 22
type_abbrev_bandlim_state, 61
type_abbrev_byte, 21
type_abbrev_duration, 19
type_abbrev_routing_table, 61
type_abbrev_tcp_seq_foreign, 22
type_abbrev_tcp_seq_local, 22
type_abbrev_tracerecord, 62
type_abbrev_ts_seq, 23

UDP_Sock, 59
UDP_Sock0, 59
udp_sock_of, 59
udp_socket, 58
udpDatagram, 27
UDPpayloadMax, 70
unix_arch, 79
update_idle, 119
update_rtt, 98
upper_timer, 47
urgent, 39

windows_arch, 79