Technical Report

UCAM-CL-TR-742
ISSN 1476-2986

Number 742

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

TCP, UDP, and Sockets: Volume 3: The Service-level Specification

Thomas Ridge, Michael Norrish, Peter Sewell

February 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

TCP, UDP, and Sockets:

Volume 3: The Service-level Specification

Thomas Ridge*
Michael Norrish†
Peter Sewell*

*University of Cambridge Computer Laboratory
†NICTA, Canberra

February 22, 2009

Abstract

Despite more than 30 years of research on protocol specification, the major protocols deployed in the Internet, such as TCP, are described only in informal prose RFCs and executable code. In part this is because the scale and complexity of these protocols makes them challenging targets for formal descriptions, and because techniques for mathematically rigorous (but appropriately loose) specification are not in common use.

In this work we show how these difficulties can be addressed. We develop a high-level specification for TCP and the Sockets API, describing the byte-stream service that TCP provides to users, expressed in the formalised mathematics of the HOL proof assistant. This complements our previous low-level specification of the protocol internals, and makes it possible for the first time to state what it means for TCP to be correct: that the protocol implements the service. We define a precise abstraction function between the models and validate it by testing, using verified testing infrastructure within HOL. Some errors may remain, of course, especially as our resources for testing were limited, but it would be straightforward to use the method on a larger scale. This is a pragmatic alternative to full proof, providing reasonable confidence at a relatively low entry cost.

Together with our previous validation of the low-level model, this shows how one can rigorously tie together concrete implementations, low-level protocol models, and specifications of the services they claim to provide, dealing with the complexity of real-world protocols throughout.

Similar techniques should be applicable, and even more valuable, in the design of new protocols (as we illustrated elsewhere, for a MAC protocol for the SWIFT optically switched network). For TCP and Sockets, our specifications had to capture the historical complexities, whereas for a new protocol design, such specification and testing can identify unintended complexities at an early point in the design.

Brief Contents

Brief Contents iii

Full Contents vii

1 Introduction to the service-level specification 3

2 Host types 19

3 Stream types 27

4 Host LTS labels and rule categories 31

5 Auxiliary functions 35

6 Auxiliary functions for TCP segment creation and drop 45

7 Host LTS: Socket Calls 57

8 Host LTS: TCP Input Processing 207
- *deliver_in_1* . . . 208
- *deliver_in_2* . . . 210
- *deliver_in_3* . . . 212
- *deliver_in_3b* . . . 220
- *deliver_in_4* . . . 221
- *deliver_in_5* . . . 221
- *deliver_in_7* . . . 222
- *deliver_in_7a* . . . 223
- *deliver_in_7b* . . . 224
- *deliver_in_7c* . . . 224
- *deliver_in_7d* . . . 225
- *deliver_in_8* . . . 226
- *deliver_in_9* . . . 226

9 Host LTS: TCP Output 229
- *deliver_out_1* . . . 230

10 Host LTS: TCP Timers 233
- *timer_tt_rexmtsyn_1* . . . 233
- *timer_tt_rexmt_1* . . . 234
- *timer_tt_persist_1* . . . 236
- *timer_tt_keep_1* . . . 236
- *timer_tt_2msl_1* . . . 237
- *timer_tt_delack_1* . . . 237
- *timer_tt_conn_est_1* . . . 237
- *timer_tt_fin_wait_2_1* . . . 238

11 Host LTS: UDP Input Processing 239
- *deliver_in_udp_1* . . . 239
- *deliver_in_udp_2* . . . 240
- *deliver_in_udp_3* . . . 240

12 Host LTS: ICMP Input Processing 241
- *deliver_in_icmp_1* . . . 241

– *deliver_in_icmp_2* . . . 242
– *deliver_in_icmp_3* . . . 244
– *deliver_in_icmp_4* . . . 245
– *deliver_in_icmp_5* . . . 246
– *deliver_in_icmp_6* . . . 246
– *deliver_in_icmp_7* . . . 247

13 Host LTS: Network Input and Output **249**
– *deliver_in_99* . . . 249
– *deliver_in_99a* . . . 250
– *deliver_out_99* . . . 250
– *deliver_loop_99* . . . 250

14 Host LTS: BSD Trace Records and Interface State Changes **253**

15 Host LTS: Time Passage **255**

16 Stream auxiliary functions **261**

17 Network labelled transition system **269**

18 Abstraction function **275**

Index **282**

Full Contents

Brief Contents iii

Full Contents vii

I Overview 1

1 Introduction to the service-level specification 3
1.1 Introduction . . . 3
1.2 Background: our previous low-level protocol model . . . 4
– *send_3* . . . 5
1.3 The new service-level specification . . . 5
– *tcpStream* . . . 6
– *streamFlags* . . . 6
– *write* . . . 6
– *send_3* . . . 7
1.4 The abstraction function . . . 7
1.5 Experimental validation . . . 8
– *abs_hosts_one_sided* . . . 9
1.6 Related work . . . 11
1.7 How to read the service-level specification . . . 12
1.8 Project History . . . 13
1.9 Conclusion . . . 14

II TCP3_hostTypes 17

2 Host types 19
2.1 The TCP control block (TCP only) . . . 19
2.1.1 Summary . . . 19
2.1.2 Rules . . . 19
– *tcpcb* . . . 19
2.2 Sockets (TCP and UDP) . . . 19
2.2.1 Summary . . . 19
2.2.2 Rules . . . 20
– *tcp_socket* . . . 20
– *protocol_info* . . . 20
– *socket* . . . 20
– *TCP_Sock0* . . . 20
– *TCP_Sock* . . . 20
– *UDP_Sock0* . . . 20
– *UDP_Sock* . . . 20
– *Sock* . . . 20
– *tcp_sock_of* . . . 20
– *udp_sock_of* . . . 20
– *proto_of* . . . 20
– *proto_eq* . . . 20

2.3 The host (TCP and UDP) . . . 21
2.3.1 Summary . . . 21
2.3.2 Rules . . . 21
– *host* . . . 21
– *privileged_ports* . . . 21
– *ephemeral_ports* . . . 21
2.4 Trace records (TCP and UDP) . . . 22
2.4.1 Summary . . . 22
2.4.2 Rules . . . 22
– *type_abbrev_tracerecord* . . . 22
– *tracecb_eq* . . . 22
– *tracesock_eq* . . . 22

III TCP3_streamTypes **25**

3 Stream types **27**
3.1 Stream types (TCP and UDP) . . . 27
3.1.1 Summary . . . 27
3.1.2 Rules . . . 27
– *type_abbrev_streamid* . . . 27
– *streamFlags* . . . 27
– *tcpStream* . . . 27
– *tcpStreams* . . . 28

IV TCP3_host0 **29**

4 Host LTS labels and rule categories **31**
4.1 Transition labels (TCP and UDP) . . . 31
4.1.1 Summary . . . 31
4.1.2 Rules . . . 31
– *Lhost0* . . . 31

V TCP3_auxFns **33**

5 Auxiliary functions **35**
5.1 Stream versions of routing functions (TCP and UDP) . . . 35
5.1.1 Summary . . . 35
5.1.2 Rules . . . 35
– *stream_test_outroute* . . . 35
– *stream_loopback_on_wire* . . . 35
5.2 Files, file descriptors, and sockets (TCP and UDP) . . . 35
5.2.1 Summary . . . 36
5.2.2 Rules . . . 36
– *sane_socket* . . . 36
5.3 Binding (TCP and UDP) . . . 36
5.3.1 Summary . . . 36
5.3.2 Rules . . . 36
– *bound_ports_protocol_autobind* . . . 36
– *bound_port_allowed* . . . 36
– *autobind* . . . 37
– *bound_after* . . . 37
– *match_score* . . . 37
– *lookup_udp* . . . 38
– *tcp_socket_best_match* . . . 38
– *lookup_icmp* . . . 39
5.4 TCP Options (TCP only) . . . 39

5.4.1 Summary . . . 39
5.4.2 Rules . . . 39
– *do_tcp_options* . . . 40
– *calculate_tcp_options_len* . . . 40
5.5 Buffers, windows, and queues (TCP and UDP) . . . 40
5.5.1 Summary . . . 40
5.5.2 Rules . . . 40
– *calculate_buf_sizes* . . . 40
– *send_queue_space* . . . 41
5.6 UDP support (UDP only) . . . 41
5.6.1 Summary . . . 41
5.6.2 Rules . . . 42
– *dosend* . . . 42
5.7 Path MTU Discovery (TCP only) . . . 42
5.7.1 Summary . . . 42
5.7.2 Rules . . . 42
– *next_smaller* . . . 42
– *mtu_tab* . . . 42
5.8 The initial TCP control block (TCP only) . . . 43
5.8.1 Summary . . . 43
5.8.2 Rules . . . 43
– *initial_cb* . . . 43

6 Auxiliary functions for TCP segment creation and drop 45
6.1 General Segment Creation (TCP only) . . . 45
6.1.1 Summary . . . 45
6.1.2 Rules . . . 45
– *tcp_output_required* . . . 45
– *tcp_output_really* . . . 45
– *stream_tcp_output_really* . . . 47
– *tcp_output_perhaps* . . . 48
– *stream_tcp_output_perhaps* . . . 48
6.2 Segment Queueing (TCP only) . . . 48
6.2.1 Summary . . . 48
6.2.2 Rules . . . 48
– *rollback_tcp_output* . . . 48
– *stream_rollback_tcp_output* . . . 49
– *enqueue_or_fail* . . . 50
– *stream_enqueue_or_fail* . . . 50
– *stream_enqueue_or_fail_sock* . . . 50
– *enqueue_and_ignore_fail* . . . 50
– *enqueue_each_and_ignore_fail* . . . 50
– *stream_mlift_tcp_output_perhaps_or_fail* . . . 50
6.3 Incoming Segment Functions (TCP only) . . . 51
6.3.1 Summary . . . 51
6.3.2 Rules . . . 51
– *update_idle* . . . 51
6.4 Drop Segment Functions (TCP only) . . . 51
6.4.1 Summary . . . 51
6.4.2 Rules . . . 51
– *dropwithreset* . . . 51
– *stream_mlift_dropafterack_or_fail* . . . 52
6.5 Close Functions (TCP only) . . . 52
6.5.1 Summary . . . 52
6.5.2 Rules . . . 52
– *tcp_close* . . . 52
– *tcp_drop_and_close* . . . 53
6.6 Socket quad testing and extraction (TCP only) . . . 53

6.6.1 Summary . . . 53
6.6.2 Rules . . . 53
– *exists_quad_of* . . . 53
– *quad_of* . . . 53

VI TCP3_hostLTS **55**

7 Host LTS: Socket Calls **57**
7.1 *accept*() (TCP only) . . . 57
7.1.1 Errors . . . 57
7.1.2 Common cases . . . 58
7.1.3 API . . . 58
7.1.4 Model details . . . 58
7.1.5 Summary . . . 59
7.1.6 Rules . . . 60
– *accept_1* . . . 60
– *accept_2* . . . 61
– *accept_3* . . . 61
– *accept_4* . . . 62
– *accept_5* . . . 63
– *accept_6* . . . 63
– *accept_7* . . . 64
7.2 *bind*() (TCP and UDP) . . . 64
7.2.1 Errors . . . 65
7.2.2 Common cases . . . 66
7.2.3 API . . . 66
7.2.4 Model details . . . 66
7.2.5 Summary . . . 67
7.2.6 Rules . . . 67
– *bind_1* . . . 67
– *bind_2* . . . 68
– *bind_3* . . . 68
– *bind_5* . . . 69
– *bind_7* . . . 69
– *bind_9* . . . 70
7.3 *close*() (TCP and UDP) . . . 70
7.3.1 Errors . . . 71
7.3.2 Common cases . . . 71
7.3.3 API . . . 72
7.3.4 Model details . . . 72
7.3.5 Summary . . . 72
7.3.6 Rules . . . 72
– *close_1* . . . 72
– *close_2* . . . 73
– *close_3* . . . 74
– *close_4* . . . 75
– *close_5* . . . 76
– *close_6* . . . 77
– *close_7* . . . 77
– *close_8* . . . 78
– *close_10* . . . 80
7.4 *connect*() (TCP and UDP) . . . 80
7.4.1 Errors . . . 81
7.4.2 Common cases . . . 82
7.4.3 API . . . 82
7.4.4 Model details . . . 83
7.4.5 Summary . . . 83
7.4.6 Rules . . . 84

– *connect_1* . . . 84
– *connect_1a* . . . 86
– *connect_2* . . . 89
– *connect_3* . . . 89
– *connect_4* . . . 90
– *connect_4a* . . . 91
– *connect_5* . . . 91
– *connect_5a* . . . 92
– *connect_5b* . . . 93
– *connect_5c* . . . 94
– *connect_5d* . . . 95
– *connect_6* . . . 95
– *connect_7* . . . 96
– *connect_8* . . . 97
– *connect_9* . . . 97
– *connect_10* . . . 98
7.5 *disconnect*() (TCP and UDP) . . . 99
7.5.1 Errors . . . 99
7.5.2 Common cases . . . 100
7.5.3 API . . . 100
7.5.4 Summary . . . 100
7.5.5 Rules . . . 100
– *disconnect_4* . . . 100
– *disconnect_5* . . . 101
– *disconnect_1* . . . 102
– *disconnect_2* . . . 103
– *disconnect_3* . . . 103
7.6 *dup*() (TCP and UDP) . . . 104
7.6.1 Errors . . . 104
7.6.2 Common cases . . . 104
7.6.3 API . . . 104
7.6.4 Summary . . . 105
7.6.5 Rules . . . 105
– *dup_1* . . . 105
– *dup_2* . . . 105
7.7 *dupfd*() (TCP and UDP) . . . 106
7.7.1 Errors . . . 106
7.7.2 Common cases . . . 106
7.7.3 API . . . 106
7.7.4 Model details . . . 106
7.7.5 Summary . . . 107
7.7.6 Rules . . . 107
– *dupfd_1* . . . 107
– *dupfd_3* . . . 107
– *dupfd_4* . . . 108
7.8 *getfileflags*() (TCP and UDP) . . . 108
7.8.1 Errors . . . 108
7.8.2 Common cases . . . 109
7.8.3 API . . . 109
7.8.4 Model details . . . 109
7.8.5 Summary . . . 109
7.8.6 Rules . . . 109
– *getfileflags_1* . . . 109
7.9 *getifaddrs*() (TCP and UDP) . . . 110
7.9.1 Errors . . . 110
7.9.2 Common cases . . . 110
7.9.3 API . . . 110
7.9.4 Model details . . . 111

7.9.5 Summary . . . 111
7.9.6 Rules . . . 111
– *getifaddrs_1* . . . 111
7.10 *getpeername*() (TCP and UDP) . . . 111
7.10.1 Errors . . . 112
7.10.2 Common cases . . . 112
7.10.3 API . . . 112
7.10.4 Model details . . . 112
7.10.5 Summary . . . 113
7.10.6 Rules . . . 114
– *getpeername_1* . . . 114
– *getpeername_2* . . . 114
7.11 *getsockbopt*() (TCP and UDP) . . . 115
7.11.1 Errors . . . 116
7.11.2 Common cases . . . 116
7.11.3 API . . . 116
7.11.4 Model details . . . 116
7.11.5 Summary . . . 116
7.11.6 Rules . . . 117
– *getsockbopt_1* . . . 117
– *getsockbopt_2* . . . 117
7.12 *getsockerr*() (TCP and UDP) . . . 118
7.12.1 Errors . . . 118
7.12.2 Common cases . . . 118
7.12.3 API . . . 118
7.12.4 Model details . . . 119
7.12.5 Summary . . . 119
7.12.6 Rules . . . 119
– *getsockerr_1* . . . 119
– *getsockerr_2* . . . 119
7.13 *getsocklistening*() (TCP and UDP) . . . 120
7.13.1 Errors . . . 120
7.13.2 Common cases . . . 120
7.13.3 API . . . 120
7.13.4 Model details . . . 121
7.13.5 Summary . . . 121
7.13.6 Rules . . . 121
– *getsocklistening_1* . . . 121
– *getsocklistening_3* . . . 122
– *getsocklistening_2* . . . 122
7.14 *getsockname*() (TCP and UDP) . . . 123
7.14.1 Errors . . . 123
7.14.2 Common cases . . . 123
7.14.3 API . . . 123
7.14.4 Model details . . . 124
7.14.5 Summary . . . 124
7.14.6 Rules . . . 124
– *getsockname_1* . . . 124
– *getsockname_2* . . . 125
– *getsockname_3* . . . 125
7.15 *getsocknopt*() (TCP and UDP) . . . 126
7.15.1 Errors . . . 126
7.15.2 Common cases . . . 126
7.15.3 API . . . 126
7.15.4 Model details . . . 127
7.15.5 Summary . . . 127
7.15.6 Rules . . . 127
– *getsocknopt_1* . . . 127

– *getsocknopt_4* . . . 128
7.16 *getsocktopt*() (TCP and UDP) . . . 128
7.16.1 Errors . . . 129
7.16.2 Common cases . . . 129
7.16.3 API . . . 129
7.16.4 Model details . . . 129
7.16.5 Summary . . . 129
7.16.6 Rules . . . 130
– *getsocktopt_1* . . . 130
– *getsocktopt_4* . . . 130
7.17 *listen*() (TCP only) . . . 131
7.17.1 Errors . . . 131
7.17.2 Common cases . . . 131
7.17.3 API . . . 131
7.17.4 Model details . . . 132
7.17.5 Summary . . . 132
7.17.6 Rules . . . 132
– *listen_1* . . . 132
– *listen_1b* . . . 133
– *listen_1c* . . . 134
– *listen_2* . . . 134
– *listen_3* . . . 135
– *listen_4* . . . 136
– *listen_5* . . . 136
– *listen_7* . . . 137
7.18 *recv*() (TCP only) . . . 137
7.18.1 Errors . . . 138
7.18.2 Common cases . . . 138
7.18.3 API . . . 138
7.18.4 Model details . . . 139
7.18.5 Summary . . . 139
7.18.6 Rules . . . 139
– *recv_1* . . . 139
– *recv_2* . . . 141
– *recv_3* . . . 142
– *recv_4* . . . 143
– *recv_7* . . . 144
– *recv_8* . . . 144
– *recv_8a* . . . 145
– *recv_9* . . . 146
7.19 *recv*() (UDP only) . . . 147
7.19.1 Errors . . . 147
7.19.2 Common cases . . . 148
7.19.3 API . . . 148
7.19.4 Model details . . . 149
7.19.5 Summary . . . 149
7.19.6 Rules . . . 150
– *recv_11* . . . 150
– *recv_12* . . . 151
– *recv_13* . . . 152
– *recv_14* . . . 152
– *recv_15* . . . 153
– *recv_16* . . . 153
– *recv_17* . . . 154
– *recv_20* . . . 155
– *recv_21* . . . 156
– *recv_22* . . . 157
– *recv_23* . . . 157

– *recv_24* . . . 158
7.20 *send*() (TCP only) . . . 159
7.20.1 Errors . . . 159
7.20.2 Common cases . . . 160
7.20.3 API . . . 160
7.20.4 Model details . . . 160
7.20.5 Summary . . . 161
7.20.6 Rules . . . 161
– *send_1* . . . 161
– *send_2* . . . 163
– *send_3* . . . 163
– *send_3a* . . . 164
– *send_4* . . . 165
– *send_5* . . . 166
– *send_5a* . . . 166
– *send_6* . . . 167
– *send_7* . . . 167
– *send_8* . . . 168
7.21 *send*() (UDP only) . . . 169
7.21.1 Errors . . . 169
7.21.2 Common cases . . . 170
7.21.3 API . . . 170
7.21.4 Model details . . . 171
7.21.5 Summary . . . 172
7.21.6 Rules . . . 173
– *send_9* . . . 173
– *send_10* . . . 174
– *send_11* . . . 175
– *send_12* . . . 176
– *send_13* . . . 177
– *send_14* . . . 178
– *send_15* . . . 179
– *send_16* . . . 179
– *send_17* . . . 180
– *send_18* . . . 181
– *send_19* . . . 182
– *send_21* . . . 182
– *send_22* . . . 183
– *send_23* . . . 184
7.22 *setfileflags*() (TCP and UDP) . . . 184
7.22.1 Errors . . . 185
7.22.2 Common cases . . . 185
7.22.3 API . . . 185
7.22.4 Model details . . . 185
7.22.5 Summary . . . 185
7.22.6 Rules . . . 185
– *setfileflags_1* . . . 185
7.23 *setsockbopt*() (TCP and UDP) . . . 186
7.23.1 Errors . . . 186
7.23.2 Common cases . . . 187
7.23.3 API . . . 187
7.23.4 Model details . . . 187
7.23.5 Summary . . . 187
7.23.6 Rules . . . 187
– *setsockbopt_1* . . . 187
– *setsockbopt_2* . . . 188
7.24 *setsocknopt*() (TCP and UDP) . . . 189
7.24.1 Errors . . . 189

7.24.2 Common cases . . . 189
7.24.3 API . . . 189
7.24.4 Model details . . . 190
7.24.5 Summary . . . 190
7.24.6 Rules . . . 190
– *setsocknopt_1* . . . 190
– *setsocknopt_2* . . . 191
– *setsocknopt_4* . . . 191
7.25 *setsocktopt*() (TCP and UDP) . . . 192
7.25.1 Errors . . . 192
7.25.2 Common cases . . . 192
7.25.3 API . . . 192
7.25.4 Model details . . . 193
7.25.5 Summary . . . 193
7.25.6 Rules . . . 193
– *setsocktopt_1* . . . 193
– *setsocktopt_4* . . . 194
– *setsocktopt_5* . . . 194
7.26 *shutdown*() (TCP and UDP) . . . 195
7.26.1 Errors . . . 195
7.26.2 Common cases . . . 195
7.26.3 API . . . 195
7.26.4 Model details . . . 196
7.26.5 Summary . . . 196
7.26.6 Rules . . . 196
– *shutdown_1* . . . 196
– *shutdown_2* . . . 197
– *shutdown_3* . . . 198
– *shutdown_4* . . . 198
7.27 socket() (TCP and UDP) . . . 199
7.27.1 Errors . . . 199
7.27.2 Common cases . . . 199
7.27.3 API . . . 199
7.27.4 Model details . . . 200
7.27.5 Summary . . . 200
7.27.6 Rules . . . 200
– *socket_1* . . . 200
– *socket_2* . . . 201
7.28 Miscellaneous (TCP and UDP) . . . 201
7.28.1 Errors . . . 202
7.28.2 Summary . . . 202
7.28.3 Rules . . . 202
– *return_1* . . . 202
– *badf_1* . . . 203
– *notsock_1* . . . 203
– *intr_1* . . . 203
– *resourcefail_1* . . . 204
– *resourcefail_2* . . . 205

8 Host LTS: TCP Input Processing **207**
8.1 Input Processing (TCP only) . . . 207
8.1.1 Summary . . . 207
8.1.2 Rules . . . 208
– *deliver_in_1* . . . 208
– *deliver_in_2* . . . 210
– *deliver_in_3* . . . 212
– *di3_topstuff* . . . 214
– *di3_newackstuff* . . . 214

– *di3_ackstuff* . . . 215
– *di3_datastuff* . . . 216
– *di3_ststuff* . . . 216
– *di3_socks_update* . . . 219
– *deliver_in_3b* . . . 220
– *deliver_in_4* . . . 221
– *deliver_in_5* . . . 221
– *deliver_in_7* . . . 222
– *deliver_in_7a* . . . 223
– *deliver_in_7b* . . . 224
– *deliver_in_7c* . . . 224
– *deliver_in_7d* . . . 225
– *deliver_in_8* . . . 226
– *deliver_in_9* . . . 226

9 Host LTS: TCP Output **229**
9.1 Output (TCP only) . . . 229
9.1.1 Summary . . . 230
9.1.2 Rules . . . 230
– *deliver_out_1* . . . 230

10 Host LTS: TCP Timers **233**
10.1 Timers (TCP only) . . . 233
10.1.1 Summary . . . 233
10.1.2 Rules . . . 233
– *timer_tt_rexmtsyn_1* . . . 233
– *timer_tt_rexmt_1* . . . 234
– *timer_tt_persist_1* . . . 236
– *timer_tt_keep_1* . . . 236
– *timer_tt_2msl_1* . . . 237
– *timer_tt_delack_1* . . . 237
– *timer_tt_conn_est_1* . . . 237
– *timer_tt_fin_wait_2_1* . . . 238

11 Host LTS: UDP Input Processing **239**
11.1 Input Processing (UDP only) . . . 239
11.1.1 Summary . . . 239
11.1.2 Rules . . . 239
– *deliver_in_udp_1* . . . 239
– *deliver_in_udp_2* . . . 240
– *deliver_in_udp_3* . . . 240

12 Host LTS: ICMP Input Processing **241**
12.1 Input Processing (ICMP only) . . . 241
12.1.1 Summary . . . 241
12.1.2 Rules . . . 241
– *deliver_in_icmp_1* . . . 241
– *deliver_in_icmp_2* . . . 242
– *deliver_in_icmp_3* . . . 244
– *deliver_in_icmp_4* . . . 245
– *deliver_in_icmp_5* . . . 246
– *deliver_in_icmp_6* . . . 246
– *deliver_in_icmp_7* . . . 247

13 Host LTS: Network Input and Output **249**
13.1 Input and Output (Network only) . . . 249
13.1.1 Summary . . . 249
13.1.2 Rules . . . 249
– *deliver_in_99* . . . 249

– *deliver_in_99a* . . . 250
– *deliver_out_99* . . . 250
– *deliver_loop_99* . . . 250

14 Host LTS: BSD Trace Records and Interface State Changes **253**
14.1 Trace Records and Interface State Changes (BSD only) . . . 253
14.1.1 Summary . . . 253
14.1.2 Rules . . . 253
– *trace_1* . . . 253
– *trace_2* . . . 253
– *interface_1* . . . 254

15 Host LTS: Time Passage **255**
15.1 Time Passage auxiliaries (TCP and UDP) . . . 255
15.1.1 Summary . . . 255
15.1.2 Rules . . . 255
– *Time_Pass_timedoption* . . . 255
– *Time_Pass_tcpcb* . . . 256
– *Time_Pass_socket* . . . 256
– *fmap_every* . . . 256
– *fmap_every_pred* . . . 256
– *Time_Pass_host* . . . 256
– *sowriteable* . . . 257
– *soreadable* . . . 258

VII TCP3_stream **259**

16 Stream auxiliary functions **261**
16.1 Default initial values (TCP and UDP) . . . 261
16.1.1 Summary . . . 261
16.1.2 Rules . . . 261
– *initial_streamFlags* . . . 261
– *initial_stream* . . . 261
– *initial_streams* . . . 262
– *streamid_of_quad* . . . 262
16.2 Auxiliary functions (TCP and UDP) . . . 262
16.2.1 Summary . . . 262
16.2.2 Rules . . . 262
– *null_flgs_data* . . . 262
– *make_syn_flgs_data* . . . 262
– *make_syn_ack_flgs_data* . . . 263
– *sync_streams* . . . 263
– *write* . . . 263
– *read* . . . 263
16.3 Stream removal (TCP and UDP) . . . 264
16.3.1 Summary . . . 264
16.3.2 Rules . . . 264
– *both_streams_destroyed* . . . 264
– *remove_destroyed_streams* . . . 264
– *destroy* . . . 264
– *destroy_quads* . . . 265

VIII TCP3_net **267**

17 Network labelled transition system **269**
17.1 Basic network types (TCP and UDP) . . . 269
17.1.1 Summary . . . 269

17.1.2 Rules . . . 269
– *type_abbrev_hosts* . . . 269
– *type_abbrev_streams* . . . 269
– *type_abbrev_msgs* . . . 269
– *type_abbrev_net* . . . 269
– *Lnet0* . . . 269
– *rn* . . . 270
17.2 Network labelled transition system (TCP and UDP) . . . 270
17.2.1 Summary . . . 270
17.2.2 Rules . . . 270
– *call* . . . 270
– *return* . . . 270
– *tau* . . . 270
– *interface* . . . 271
– *host_tau* . . . 271
– *time_pass* . . . 271
– *trace* . . . 271

IX TCP3_absFun **273**

18 Abstraction function **275**
18.1 Auxiliary functions (TCP and UDP) . . . 275
18.1.1 Summary . . . 275
18.1.2 Rules . . . 275
– *tcpcb1_to_3* . . . 275
– *tcp_socket1_to_3* . . . 275
– *socket1_to_3* . . . 275
– *host1_to_3* . . . 276
18.2 Stream reassembly (TCP and UDP) . . . 276
18.2.1 Summary . . . 276
18.2.2 Rules . . . 276
– *stream_reass* . . . 276
18.3 Abstraction function (TCP and UDP) . . . 277
18.3.1 Summary . . . 277
18.3.2 Rules . . . 277
– *ERROR* . . . 277
– *abs_hosts_one_sided* . . . 277
– *abs_hosts* . . . 278
– *abs_lbl* . . . 279
– *abs_trans* . . . 279

Index **282**

Part I

Overview

Chapter 1

Introduction to the service-level specification

1.1 Introduction

Real-world network protocols are usually described in informal prose RFCs, which inevitably have unintentional ambiguities and omissions, and which do not support conformance testing, verification of implementations, or verification of applications that use these protocols. Moreover, there are many subtly different realisations, including the TCP implementations in BSD, Linux, WinXP, and so on. The Internet protocols have been extremely successful, but the cost is high: there is considerable legacy complexity that implementors and users have to deal with, and there is no clear point of reference. To address this, we have developed techniques to put practical protocol design on a rigorous footing, to make it possible to specify protocols and services with mathematical precision, and to do verified conformance testing directly against those specifications. In this work we demonstrate our approach by developing and validating a high-level specification of the service provided by TCP: the dominant data transport protocol (underlying email and the web), which provides reliable duplex byte streams, with congestion control, above the unreliable IP layer.

Our specification deals with the full complexity of the service provided by TCP (except for performance properties). It includes the Sockets API (`connect`, `listen`, etc.), hosts, threads, network interfaces, the interaction with ICMP and UDP, abandoned connections, transient and persistent connection problems, unexpected socket closure, socket self-connection and so on. The specification comprises roughly 30 000 lines of (commented) higher-order logic, and mechanized tool support has been essential for work on this scale. It is written using the HOL system [14]. The bulk of the definition is an operational semantics, using idioms for timed transition relations, record-structured state, pattern matching and so on.

We relate this service-level specification to our previous protocol description by defining, again in HOL, an abstraction function from the (rather complex) low-level protocol states, with sets of TCP segments on the wire, flow and congestion control data, etc., to the (simpler) service-level states, comprising byte streams and some status information. This makes explicit how the protocol implements the service.

The main novelty of the approach we take here is the *validation* of this abstraction function. Ideally, one would *prove* that the abstraction relationship holds in all reachable states. Given the scale and complexity of the specifications, however, it is unclear whether that would be pragmatically feasible, especially with the limited resources of an academic team. Accordingly, we show how one can validate the relationship by verified testing. We take traces of the protocol-level specification (themselves validated against the behaviour of the BSD TCP implementation), and verify (automatically, and in HOL) that there are corresponding traces of the service-level specification, with the abstraction function holding at each point. Our previous protocol-level validation, using a special-purpose symbolic evaluator, produced symbolic traces of the protocol-level specification. We now *ground* these traces, using a purpose-built constraint solver to instantiate variables to satisfy any outstanding constraints, and use a new symbolic evaluator to apply the abstraction function and check that the resulting trace lies in the service-level specification. By doing this all within HOL, we have high confidence in the validation process itself.

Obviously, such testing cannot provide complete guarantees, but our experience with the kind of errors it detects suggests that it is still highly discriminating (partly due to the fact that it examines the internal states of the specifications at every step along a trace) and one can develop useful levels of confidence relatively quickly.

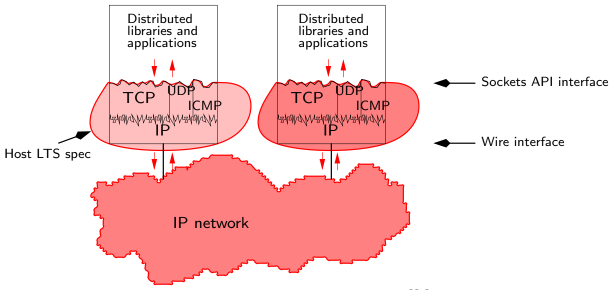
In the following sections, we first recall our previous protocol model (Sect. 1.2), before describing the new service-level specification (Sect. 1.3) and abstraction function (Sect. 1.4), giving small excerpts from each. We then discuss the validation infrastructure, and the results of validation (Sect. 1.5). Finally, we discuss related work and conclude.

1.2 Background: our previous low-level protocol model

Our previous low-level specification [5, 8] characterises TCP, UDP and ICMP at the protocol level, including hosts, threads, the Sockets API, network interfaces and segments on the wire. As well as the core functionality of segment retransmission and flow control, TCP must handle details of connection setup and tear-down, window scaling, congestion control, timeouts, optional TCP features negotiated at connection setup, interaction with ICMP messages, and so on. The model covers all these. It is parameterized by the OS, allowing OS-dependent behaviour to be specified cleanly; it is also non-deterministic, so as not to constrain implementations unnecessarily.

This level of detail results in a model of roughly 30 000 lines of (commented) higher-order logic (similar in size to the implementations, but structured rather differently). As further evidence of its accuracy and completeness, it has been successfully used as the basis for a Haskell implementation of a network stack [17].

The main part of the protocol model (the pale shaded region below) is the *host labelled transition system*, or *host LTS*, describing the possible interactions of a host OS: between program threads and host via calls and returns of the Sockets API, and between host and network via message sends and receives. The protocol model uses the host LTS, and a model of the TCP, UDP and ICMP segments on the wire, to describe a network of communicating hosts.



The host labelled transition relation, $h \xrightarrow{lbl} h'$, is defined by some 148 rules for the socket calls (5–10 for each interesting call) and some 46 rules for message send/receive and for internal behaviour. An example of one of the simplest rules is given in Fig. 1.1. The rule describes a host with a blocked thread attempting to send data to a socket. The thread becomes unblocked and transfers the data to the socket's send queue. The send call then returns to the user.

The transition $h \, (\!|...|\!) \xrightarrow{\tau} h \, (\!|...|\!)$ appears at the top, where the thread pointed to by *tid* and the socket pointed to by *sid* are unpacked from the original and final hosts, along with the send queue *sndq* for the socket. Host fields that are modified in the transition are highlighted. The initial host has thread *tid* in state SEND2, blocking attempting to send *str* to *sndq*. After the transition, *tid* is in state RET(OK...), about to return to the user with *str''*, the data that has not been sent, here constrained to be the empty string.

The bulk of the rule is the condition (a predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. The condition is simply a conjunction of clauses, with no temporal ordering. The rule only applies if the state of the socket, *st*, is either ESTABLISHED or CLOSE_WAIT. Then, provided send_queue_space is large enough, *str* is appended to the *sndq* in the final host. Lastly, the urgent pointer *sndurp'* is set appropriately.

Although the bulk of the model deals with the relatively simple Sockets API, with many rules like that of Fig. 1.1, the real complexity arises from internal actions that are largely invisible to the Sockets user, such as retransmission and congestion control. For example, the rule *deliver_in_3* (not shown) that handles normal message receipt comprises over 1 000 lines of higher-order logic.

The model has been validated against several thousand real-world network traces, designed to test corner cases and unexpected situations. Of these, 92% are valid according to the model, and we believe that for many purposes the model is sufficiently accurate — certainly enough to be used as a reference, in conjunction with the standard texts.

send_3 **tcp: slow nonurgent succeed** **Successfully return from blocked state having sen t data**

$$h \, \langle\!| ts := ts \oplus (tid \mapsto \boxed{(\textsc{Send2}(sid, *, str, opts))_{d}});$$
$$socks := socks \oplus \; [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *, \boxed{sndq}, \boxed{sndurp}, rcvq, rcvurp, iobc)))] |\!\rangle$$
$$\xrightarrow{\tau}$$
$$h \, \langle\!| ts := ts \oplus (tid \mapsto \boxed{(\textsc{Ret}(\text{OK}(\textbf{implode } str'')))_{\text{sched\_timer}}});$$
$$socks := socks \oplus \; [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *, \boxed{sndq ++ str'}, \boxed{sndurp'}, rcvq, rcvurp, iobc)))] |\!\rangle$$

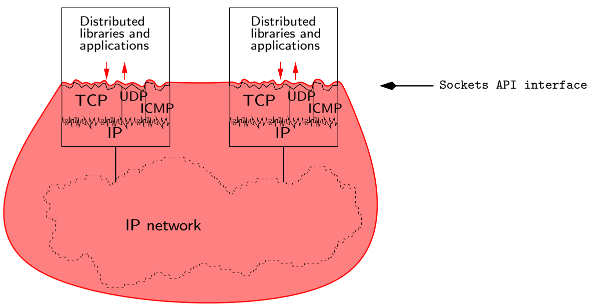
$st \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \wedge$
$space \in \text{send\_queue\_space}(sf.n(\text{SO\_SNDBUF}))$
$(\textbf{length } sndq)(\text{MSG\_OOB} \in opts)$
$h.\text{arch } cb.t\_maxseg \; i_2 \wedge$
$space \geq \textbf{length } str \wedge$
$str' = str \wedge str'' = [\,] \wedge$
$sndurp' = \textbf{if } \text{MSG\_OOB} \in opts \textbf{ then } \uparrow(\textbf{length}(sndq ++ str') - 1) \textbf{ else } sndurp$

HOL syntax For optional data items, $*$ denotes absence (or a zero IP or port) and $\uparrow x$ denotes presence of value x. Concrete lists are written $[1, 2, 3]$ and appending two lists is written using an infix $++$. Records are written within angled brackets $\langle\!| \ldots |\!\rangle$. Record fields can be accessed by dot notation or by pattern-matching. Record fields may be overridden: $cb' = cb \, \langle\!| irs := seq |\!\rangle$ states that the record cb' is the same as the record cb, except that field $cb'.irs$ has the value seq. The expression $f \oplus [(x, y)]$ or $f \oplus (x \mapsto y)$ denotes the finite map f updated to map x to y.

Figure 1.1: Protocol-level model, example rule

1.3 The new service-level specification

The service-level specification, illustrated below, describes the behaviour of a network of hosts communicating over TCP, as observed at the Socket APIs of the connections involved. It does not deal with TCP segments on the wire (though it necessarily does include ICMP and UDP messages).



In principle one could derive a service-level specification directly from the protocol model, taking the set of traces it defines and erasing the TCP wire segment transitions. However, that would not give a *usable* specification: one in which key properties of TCP, that users depend on, are clearly visible. Hence, we built the service-level specification by hand, defining a more abstract notion of host state, an abstract notion of stream object, and a new network transition relation, but aiming to give the same Sockets-API-observable behaviour.

The abstract host states are substantially simpler than those of the protocol-level model. For example, the protocol-level TCP control block contains 44 fields, including retransmit and keep-alive timers; window sizes, sequence position and scaling information; timestamping and round trip times. Almost none of these are relevant to the service-level observable behaviour, and so are not needed in the service-level TCP control block. Along with this, the transition rules that define the protocol dynamics, such as *deliver_in_3*, become much simpler. The rules that deal with the Sockets API must be adapted to the new host state, but they remain largely as before. The overall size of the specification is therefore not much changed, at around 30 000 lines (including comments).

A naive approach to writing the individual rules would be to existentially quantify those parts of the host state that are missing at the service level (and then to logically simplify as much as possible). However, this would lead to a highly non-deterministic and ultimately less useful specification. Instead, we relied on a number of invariants of the low-level model, arguing informally that, given those, the two behaviours match. We rely on the later validation to detect any errors in these informal arguments.

In the rest of this section we aim to give a flavour of the service-level specification, the details of which are included in later parts of this document.

The heart of the specification is a model of a bidirectional TCP connection as a pair of unidirectional byte streams between Sockets endpoints:

– **unidirectional stream :**

$$\begin{aligned}\mathsf{tcpStream} = \langle\!| \; & i : \mathsf{ip}; \text{ (* source IP *)}\\ & p : \mathsf{port}; \text{ (* source port *)}\\ & \mathit{flgs} : \mathsf{streamFlags};\\ & \mathit{data} : \mathsf{byte\ list};\\ & \mathit{destroyed} : \mathsf{bool} \,|\!\rangle\end{aligned}$$

The data in the stream is a byte list. Further fields record the source IP address and port of the stream, control information in the form of flags, and a boolean indicating whether the stream has been destroyed at the source (say, by deleting the associated socket). Some of these fields are shared with the low-level specification, but others are purely abstract entities. Note that although a stream may be destroyed at the source, previously sent messages may still be on the wire, and might later be accepted by the receiver, so we cannot simply remove the stream when it is destroyed. Similarly, if the source receives a message for a deleted socket, a RST will typically be generated, which must be recorded in the stream flags of the destroyed stream. These flags record whether the stream is opening (SYN,$SYNACK$), closing normally (FIN) or abnormally (RST).

– **stream control information :**

$$\begin{aligned}\mathsf{streamFlags} = \langle\!| \; & SYN : \mathsf{bool}; \text{ (* } SYN\text{, no } ACK \text{ *)}\\ & SYNACK : \mathsf{bool}; \text{ (* } SYN \text{ with } ACK \text{ *)}\\ & FIN : \mathsf{bool};\\ & RST : \mathsf{bool} \,|\!\rangle\end{aligned}$$

This control information is carefully abstracted from the protocol level, to capture just enough structure to express the user-visible behaviour. Note that the SYN and $SYNACK$ flags may be set simultaneously, indicating the presence of both kinds of message on the wire. The receiver typically lowers the stream SYN flag on receipt of a SYN: even though messages with a SYN may still be on the wire, subsequent SYNs will be detected by the receiver as invalid duplicates of the original. A bidirectional stream is then just an unordered pair (represented as a set) of unidirectional streams.

The basic operations on a byte stream are to read and write data. The following defines a write from Sockets endpoint (i_1, p_1) to endpoint (i_2, p_2).

– **write flags and data to a stream :**

$$\begin{aligned}&\mathrm{write}(i_1, p_1, i_2, p_2)(\mathit{flgs}, \mathit{data})s\ s' = (\\ &\quad \exists \mathit{in\_}\ \mathit{out}\ \mathit{in}'\ \mathit{out}'.\\ &\quad \mathrm{sync\_streams}(i_1, p_1, i_2, p_2)s(\mathit{in\_}, \mathit{out}) \wedge\\ &\quad \mathrm{sync\_streams}(i_1, p_1, i_2, p_2)s'(\mathit{in}', \mathit{out}') \wedge\\ &\quad \mathit{in}' = \mathit{in\_} \wedge\\ &\quad \mathit{out}'.\mathit{flgs} =\\ &\quad \langle\!|\ SYN := (\mathit{out}.\mathit{flgs}.SYN \vee \mathit{flgs}.SYN);\\ &\qquad SYNACK := (\mathit{out}.\mathit{flgs}.SYNACK \vee \mathit{flgs}.SYNACK);\\ &\qquad FIN := (\mathit{out}.\mathit{flgs}.FIN \vee \mathit{flgs}.FIN);\\ &\qquad RST := (\mathit{out}.\mathit{flgs}.RST \vee \mathit{flgs}.RST) |\!\rangle \wedge\\ &\quad \mathit{out}'.\mathit{data} = (\mathit{out}.\mathit{data} + +\mathit{data}))\end{aligned}$$

Stream s' is the result of writing flgs and data to stream s. Stream s consists of a unidirectional input stream $\mathit{in\_}$ and output stream out, extracted from the bidirectional stream using the auxiliary sync_streams function. Similarly s', the state of the stream after the write, consists of in' and out'. Since we are writing to the output stream, the input stream remains unchanged, $\mathit{in}' = \mathit{in\_}$. The flags on the output stream are modified to reflect flgs. For example, SYN is set in $\mathit{out}'.\mathit{flgs}$ iff flgs contains a SYN or $\mathit{out}.\mathit{flgs}$ already has SYN set. Finally, $\mathit{out}'.\mathit{data}$ is updated by appending data to $\mathit{out}.\mathit{data}$.

Fig. 1.2 gives the service-level analogue of the previous protocol-level rule. The transition occurs between triples $(h\ \langle\!|...|\!\rangle, S_0 \oplus [...], M)$, each consisting of a host, a finite map from stream identifiers to *send_3* **tcp: slow nonurgent succeed** **Successfully return from blocked state having sent data**

$$
\begin{array}{l}
(h\ \langle\!|\, ts := ts \oplus (tid \mapsto (\textsc{Send2}(sid, *, str, opts))_{d});\\
\quad socks := socks \oplus [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad \mathrm{TCP\_Sock}(st, cb, *)))] |\!\rangle,\\
S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], M)\\
\xrightarrow{\tau} (h\ \langle\!|\, ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}(\mathbf{implode}\ str'')))_{sched\_timer});\\
\quad socks := socks \oplus [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad \mathrm{TCP\_Sock}(st, cb, *)))] |\!\rangle,\\
S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], M)
\end{array}
$$

$$
\begin{array}{l}
st \in \{\mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}\} \wedge\\
space \in \mathrm{UNIV} \wedge\\
space \geq \mathbf{length}\ str \wedge\\
str' = str \wedge str'' = [\,] \wedge\\
flgs = flgs\ \langle\!|\, SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge\\
\mathrm{write}(i_1, p_1, i_2, p_2)(flgs, str')s\ s'
\end{array}
$$

Figure 1.2: Service-level specification, example rule

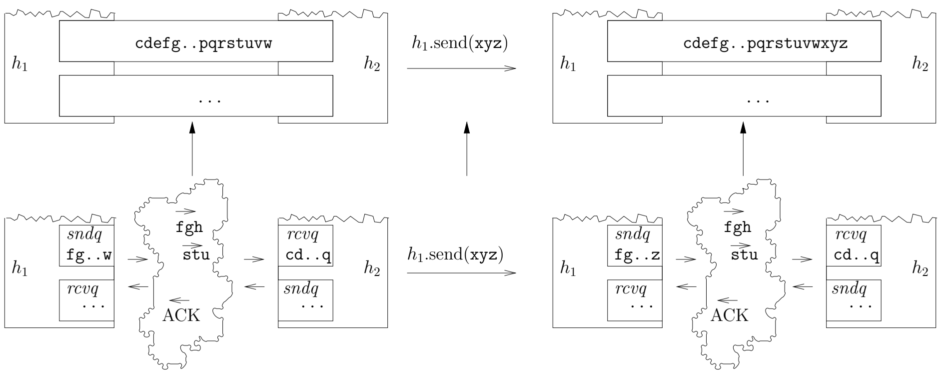


Figure 1.3: Abstraction function, illustrated (data part only)

streams, and a set of UDP and ICMP messages. The latter do not play an active part in this rule, and can be safely ignored. Host state is unpacked from the host as before. Note that protocol-level constructs such as *rcvurp* and *iobc* are absent from the service-level host state. As well as the host transition, there is a transition of the related stream s to s'. The stream is unpacked from the finite map via its unique identifier streamid_of_quad(i_1, p_1, i_2, p_2), derived from its quad.

As before, the conditions for this rule require that the state of the socket *st* must be ESTABLISHED or CLOSE_WAIT. Stream s' is the result of writing string str' and flags *flgs* to s. Since *flgs* are all false, the write does not cause any control flags to be set in s', although they may already be set in s of course.

This rule, and the preceding definitions, demonstrate the conceptual simplicity and stream-like nature of the service level. Other interesting properties of TCP are clearly captured by the service-level specification. For example, individual writes do not insert record boundaries in the byte stream, and in general, a read returns only part of the data, uncorrelated with any particular write. The model also makes clear that the unidirectional streams are to a large extent independent. For example, closing one direction does not automatically cause the other to close.

1.4 The abstraction function

While the service specification details *what* service an implementation of TCP provides to the Sockets interface, the abstraction function details *how*. The abstraction function maps protocol-level states and transitions to service-level states and transitions. A protocol-level network consists of a set of hosts, each with their own TCP stacks, and segments on the wire. The abstraction function takes this data and calculates abstract byte streams between Sockets API endpoints, together with the abstract connection status information.

The latter is the more intricate part, but we can give only a simple example here: the *destroyed* flag is set iff either there is no socket on the protocol-level host matching the quad for the TCP connection or the state of the TCP socket is CLOSED.

The former is illustrated in Fig. 1.3. For example, consider the simple case where communication has already been established, and the source is sending a message to the destination that includes the string "`abc...xyz`", of which bytes up to "`w`" have been moved to the source *sndq*. Moreover, the destination has acknowledged all bytes up to "`f`", so that the *sndq* contains "`fgh...uvw`", and *snd_una* points to "`f`". The destination *rcvq* contains "`cde...opq`", waiting for the user to read from the socket, and *rcv_nxt* points just after "`q`".

	↓ *snd_una* ↓ *rcv_nxt*
message	`...abcdefghijklmnopqrstuvwxyz...`
source *sndq*	`fghijklmnopqrstuvw`
destination *rcvq*	`cdefghijklmnopq`
DROP(*rcv_nxt* − *snd_una*)*sndq*	`rstuvw`
stream	`cdefghijklmnopqrstuvw`

The data that remains in the stream waiting for the destination endpoint to read, is the byte stream "`cdefghijklmnopqrstuvw`". This is simply the destination *rcvq* with part of the source *sndq* appended: to avoid duplicating the shared part of the byte sequence, (*rcv_nxt* − *snd_una*) bytes are dropped from *sndq* before appending it to *rcvq*.

An excerpt from the HOL definition appears in Fig. 1.4. It takes a quad (i_1, p_1, i_2, p_2) identifying the TCP connection, a source host h, a set of messages *msgs* on the wire, and a destination host i, and produces a unidirectional stream. It follows exactly the previous analysis: (*rcv_nxt* − *snd_una*) bytes are dropped from *sndq* to give *sndq*′, which is then appended to *rcvq* to give the data in the stream.

Note that, in keeping with the fact that TCP is designed so that hosts can retransmit any data that is lost on the wire, this abstraction does not depend on the data in transit — at least for normal connections in which neither endpoint has crashed.

For a given TCP connection, the full abstraction function uses the unidirectional function twice to form a bidirectional stream constituting the service-level state. As well as mapping the states, the abstraction function maps the transition labels. Labels corresponding to visible actions at the Sockets interface, such as a `connect` call, map to themselves. Labels corresponding to internal protocol actions, such as the host network interface sending and receiving datagrams from the wire, are invisible at the service level, and so are mapped to τ, indicating no observable transition. Thus, for each protocol-level transition, the abstraction function gives a service-level transition with the same behaviour at the Sockets interface. Mapping the abstraction function over a protocol-level trace gives a service-level trace with identical Sockets behaviour. Every valid protocol-level trace should map to a valid service-level trace.

1.5 Experimental validation

How can we ensure that TCP implementations (written in C), our previous protocol-level model (in HOL), and our new service-level specification (also in HOL) are consistent? Arguing that a small specification corresponds to a simple real-world system can already be extremely challenging. Here, we are faced with very large specifications and a very complex real-world system. Ideally one would verify the relationship between the protocol and service specifications by proving that their behaviours correspond, making use of the abstraction function. One would also prove that the Sockets behaviour of the endpoint implementations (formalized using a C semantics) conformed to the protocol model.

Proving the relationships between the levels in this way would be a very challenging task indeed. One of the main barriers is the scale of TCP implementations, including legacy behavioural intricacies of TCP and Sockets, which were not designed with verification in mind.

Hence, we adopt the pragmatic approach of validating the specifications to provide reasonable confidence in their accuracy. Note that for TCP the implementations are the de facto standard. In producing specifications after the fact, we aim to validate the specifications against the implementation behaviour. Our techniques could equally well be used in the other direction for new protocol designs. Our service-level validation builds on our earlier protocol-level work [5, 8], so we begin by recalling that.

Protocol-level validation We instrumented a test network and wrote tests to drive hosts on the network, generating real-world traces. We then ensured that the protocol specification admitted those traces by running a special-purpose symbolic model checker in HOL, correcting the specification, and

– **unidirectional abstraction function :**

```
abs_hosts_one_sided(i1, p1, i2, p2)(h, msgs, i) = (
  (* messages that we are interested in, including oq and iq *)
  let (hoq, iiq) =
      case (h.oq, i.iq) of ((msgs)_1 , (msgs')_2) → (msgs, msgs') in
  let msgs = list_to_set hoq ∪ msgs ∪(list_to_set iiq) in
  (* only consider TCP messages ...*)
  let msgs = {msg | TCP msg ∈ msgs} in
  (* ...that match the quad *)
  let msgs = msgs ∩
      {msg | msg = msg ⟨| is1 :=↑ i1; ps1 :=↑ p1; is2 :=↑ i2; ps2 :=↑ p2|⟩} in

  (* pick out the send and receive sockets *)
  let smatch i1 p1 i2 p2 s =
    ((s.is1, s.ps1, s.is2, s.ps2) = (↑ i1,↑ p1,↑ i2,↑ p2)) in
  let snd_sock = Punique_range(smatch i1 p1 i2 p2)h.socks in
  let rcv_sock = Punique_range(smatch i2 p2 i1 p1)i.socks in
  let tcpsock_of sock = case sock.pr of
      TCP1_hostTypes $TCP_PROTO tcpsock → tcpsock
    ‖ _3 → ERROR"abs_hosts_one_sided:tcpsock_of"
  in
  (* the core of the abstraction function is to compute data *)
  let (data : byte list) = case (snd_sock, rcv_sock) of
      (↑(_8, hsock),↑(_9, isock)) → (
          let htcpsock = tcpsock_of hsock in
          let itcpsock = tcpsock_of isock in
          let (snd_una, sndq) = (htcpsock.cb.snd_una, htcpsock.sndq) in
          let (rcv_nxt, rcvq) = (itcpsock.cb.rcv_nxt, itcpsock.rcvq) in
          let rcv_nxt = tcp_seq_flip_sense rcv_nxt in
          let sndq' = DROP((num(rcv_nxt − snd_una)))sndq in
          rcvq + +sndq')

    ‖ (↑(_8, hsock), ∗) → (
          let htcpsock = tcpsock_of hsock in
          htcpsock.sndq)

    ‖ (∗,↑(_9, isock)) → (
          let itcpsock = tcpsock_of isock in
          let (rcv_nxt : tcpLocal seq32, rcvq : byte list) =
              (tcp_seq_flip_sense(itcpsock.cb.rcv_nxt), itcpsock.rcvq) in
          rcvq + +(stream_reass rcv_nxt msgs))

    ‖ (∗, ∗) → ERROR"abs_hosts_one_sided:data"
  in
  ⟨| i := i1;
    p := p1;
    flgs :=
    ⟨| SYN :=(∃msg.msg ∈ msgs ∧ msg = msg ⟨| SYN := T; ACK := F|⟩);
      SYNACK :=(∃msg.msg ∈ msgs ∧ msg = msg ⟨| SYN := T; ACK := T|⟩);
      FIN :=(∃msg.msg ∈ msgs ∧ msg = msg ⟨| FIN := T|⟩);
      RST :=(∃msg.msg ∈ msgs ∧ msg = msg ⟨| RST := T|⟩)
    |⟩;
    data := data;
    destroyed :=(case snd_sock of
    ↑(sid, hsock) → ((tcpsock_of hsock).st = CLOSED)
     ‖ ∗ → T)
  |⟩)
```

Figure 1.4: Abstraction function, excerpt

iterating, when we discovered errors. Because it is based directly on the formal specification, and deals with all the internal state of hosts, the checker is extremely rigorous, producing a machine checked proof of admissibility for each successfully validated trace. Obviously no testing-based method can be complete, but this found many issues in early drafts of the specification, and also identified a number of anomalies in TCP implementations.

Service-level validation For the service-level validation, we began with a similar instrumented test network, but collected double-ended traces, capturing the behaviour of two interacting hosts, rather than just one endpoint. We then used our previous symbolic evaluation tool to discover symbolic traces of the protocol-level model that corresponded to the real-world traces. That is a complex and computationally intensive process, involving backtracking depth-first search and constraint simplification, essentially to discover internal host state and internal transitions that are not explicit in the trace.

We then *ground* these symbolic traces, finding instantiations of their variables that satisfy any remaining constraints, to produce a ground protocol-level trace in which all information is explicit. Given such a ground trace, we can map the abstraction function over it to produce a candidate ground service-level trace.

It is then necessary to check validity of this trace, which is done with a service-level test oracle. As at the protocol level, we wrote a new special-purpose service-level checker in HOL which performs symbolic evaluation of the specification with respect to ground service-level traces. Crucially, this checking process is much simpler than that at the protocol level because all host values, and all transitions, are already known. All that remains is to check each ground service-level transition against the specification.

The most significant difference between the old and new checkers is that the former had to perform a depth-first search to even determine which rule of the protocol model was appropriate. Because that work has already been done, and because the two specifications have been constructed so that their individual rules correspond, the service-level checker does not need to do this search. Instead, it can simply check the service-level version of the rule that was checked at the protocol level, dealing with each transition in isolation. In particular, this means that the service-level checker need not attempt to infer the existence of unobservable τ-transitions.

Another significant difference between the two checkers is that the service-level checker can aggressively search for instantiations of existentially quantified variables that arise when a rule's hypothesis has to be discharged. At the protocol level, such variables may appear quite unconstrained at first appearance, but then become progressively more constrained as further steps of the trace are processed.

For example, a simplified rule for the `socket` call might appear as

$$\frac{fd \notin \mathsf{usedfds}(h_0)}{h_0 \langle\!| socks := socks |\!\rangle \xrightarrow{tid \cdot \mathsf{socket}()} h_0 \langle\!| socks := socks \oplus (sid, fd) |\!\rangle}$$

stating that when a `socket` call is made, the host h_0's `socks` map is updated to associate the new socket (identified by *sid*) with file-descriptor *fd*, subject only to the constraint that the new descriptor not already be in use. (This under-specification is correct on Windows; on Unix, the file-descriptor is typically the next available natural number.)

In the protocol-level checker, the *fd* variable must be left uninstantiated until its value can be deduced from subsequent steps in the trace. In the service-level checker, both the initial host and the final host are available because they are the product of the abstraction function applied to the previously generated, and ground, protocol trace. In a situation such as this, the variable from the hypothesis is present in the conclusion, and can be immediately instantiated.

In other rules of the service-level specification, there can be a great many variables that occur only in the hypothesis. These are existentially quantified, and the checker must determine if there is an instantiation for them that makes the hypothesis true. The most effective way of performing this check is to simplify, apply decision procedures for arithmetic, and to then repeatedly case-split on boolean variables, and the guards of if-then-else expressions to search for possible instantiations.

The above process is clearly somewhat involved, and itself would ordinarily be prone to error. To protect against this we built all the checking infrastructure within HOL. So, when checking a trace, we are actually building machine-checked proofs that its transitions are admitted by the inductive definition of the transition relation in the specification.

Results Our earlier protocol-level validation involved several thousand traces designed to exercise the behaviour of single endpoints, covering both the Sockets API and the wire behaviour. To produce a reasonably accurate specification, we iterated the checking and specification-fixing process many times.

For the service-level specification, we have not attempted the same level of validation, simply due to resource constraints. Instead, we have focused on developing the method, doing enough validation to demonstrate its feasibility. Producing a specification in which one should have high confidence might require another man-year or so of testing — perfectly feasible, and a tiny amount of effort in terms of industrial protocol stack development, but unlikely to lead to new research insights. That said, most of the Sockets API behaviour does not relate to the protocol dynamics and is common between the two specifications, so is already moderately well tested. In all, 30 end-to-end tests were generated, covering a variety of connection setup and tear-down cases and end-to-end communication, but not including packet loss, reordering, duplication, and severe delay. After correcting errors, all these traces were found to validate successfully.

To illustrate how discriminating our testing process is, we mention two errors we discovered during validation. At the protocol-level, a TCP message moving from a host output queue to the wire corresponds to an unobservable τ event at the service level. Naively we assumed the host state would be unchanged, since the output queue at the service-level carries only ICMP and UDP messages. However, this is not correct, since the transmission of a TCP message alters the timer associated with the output queue, increasing its value. The update to the timer permits the host to delay sending the ICMP and UDP messages. Without this side-effect, the service-level specification effectively required ICMP and UDP messages to be sent earlier than they would otherwise have been. To correct this error, the service specification had to allow the timer to be updated if at the protocol-level there was potentially a TCP message on the queue that might be transferred to the wire. Another error arose in the definition of the abstraction function. The analysis of the merging of the send and receive queues on source and destination hosts, described in Sect. 1.4, was initially incorrect, leading to streams with duplicated, or missing, runs of data. Fortunately this error was easy to detect by examining the ground service-level trace, where the duplicated data was immediately apparent.

Our validation processes check that certain traces are included in the protocol-level or service-level specification. As we have seen, this can be a very discriminating test, but it does not touch on the possibility that the specifications admit too many traces. That cannot be determined by reference to the de facto standard implementations, as a reasonable specification here must be looser than any one implementation. Instead, one must consider whether the specifications are strong enough to be useful, for proving properties of applications that use the Sockets API, or (as in [17]) as a basis for new implementations.

1.6 Related work

This work builds on our previous TCP protocol model [6, 7, 5, 8], and we refer the reader there for detailed discussion of related work. We noted that "to the best of our knowledge, however, no previous work approaches a specification dealing with the full scale and complexity of a real-world TCP". This also applies to the service-level specification. As before, this is unsurprising: we have depended on automated reasoning tools and on raw compute resources that were simply unavailable in the 1980s or early 1990s. Our goals have also been different, and in some sense more modest, than the correctness theorems of traditional formal verification: we have not attempted to *prove* that an implementation of TCP satisfies the protocol model, or that the protocol satisfies the service-level specification.

Since the protocol model was published, there have been several papers extending our work in various directions. As part of his thesis on massively concurrent applications in Haskell [16], Peng Li translated the protocol specification to Haskell to produce an executable user-space TCP stack. Compton verified Stenning's protocol based on our UDP model [11]. Subsequently we verified an implementation of a persistent message queue based on a model of TCP that, although different from the service-level specification, was heavily influenced by it [23].

There is a vast literature devoted to verification techniques for protocols, with both proof-based and model checking approaches, e.g. in conferences such as CAV, CONCUR, FM, FORTE, ICNP, SPIN, and TACAS. The most detailed rigorous specification of a TCP-like protocol we are aware of is that of Smith [27], an I/O automata specification and implementation, with a proof that one satisfies the other, used as a basis for work on T/TCP. The protocol is still substantially idealised, however. Later work by Smith and Ramakrishnan uses a similar model to verify properties of a model of SACK [26]. A variety of work addresses radically idealised variants of TCP [10, 12, 24, 13, 3, 19, 20]. Finally, Postel's PhD thesis used early Petri net protocol models descriptively [22].

Implementations of TCP in high-level languages have been written by Biagioni in Standard ML [2], by Castelluccia *et al.* in Esterel [9], and by Kohler *et al.* in Prolac [15]. As for any implementation, allowable non-determinism means they cannot be used as oracles for conformance testing.

For concurrent and distributed systems, there are many abstraction-refinement techniques, such as abstraction relations (which include our abstraction function) and simulation relations, see [18] for an overview. As an example of these techniques, Alur and Wang address the PPP and DHCP protocols [1]. For each they check refinements between models that are manually extracted from the RFC specification and from an implementation. Although these techniques are widely used in verification, to the best of our knowledge, they have never been applied previously to real-world protocols on the scale of TCP.

1.7 How to read the service-level specification

This document is the third volume of a series. The first two volumes describe the protocol-level specification. For a full discussion of the protocol-level specification we refer the reader to the companion *TCP, UDP and Sockets, Volume 1: Overview* [6] 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 protocol-level specification itself is detailed in *TCP, UDP and Sockets, Volume 2: The Specification* [7].

The service-level is closely related to the protocol-level (as the abstraction function makes clear), and the two specifications are similar in many ways. For example, the service-level specification of the host transition relation closely parallels that of the protocol-level. The reader familiar with the syntax and format of the protocol-level rules should find the service-level very familiar. Therefore the overview of the protocol-level [6] is recommended as an introduction to the style and formalism employed in the service-level specification. We briefly summarize the main differences between the HOL theory files of the two specifications.

- The service-level host types in Sect. 2 are more abstract than those at the protocol-level. For example, a TCP control block contains 44 fields at the protocol level, compared with 2 at the service level.
- The formal definition of byte streams in Sect. 3 is not present at the protocol level.
- The rule labels in Sect. 4 are essentially the same as those at the protocol level. Although the rule labels match, it is worth recalling that TCP datagram sends and receives at the protocol level will be replaced by stream interactions at the service level. The service-level datagram labels are primarily used for UDP and ICMP messages.
- The auxiliary functions in Sect. 5 are similar to those at the protocol level.
- The Sockets rules in Sect. 7 correspond one-to-one with those at the protocol level. For the most part they are minor simplifications of the corresponding protocol-level rules. The Sockets API embodies considerable complexity independent of the internal functioning of TCP, which is why these rules are not much simpler.
- The interal functioning of TCP in Sect. 8 and Sect. 9 is significantly simpler than that at the protocol level, because much of the detail of TCP, such as retransmission, has been abstracted away.
- The behaviour of the byte-streams described in Sect. 16 is unique to the service-level specification.
- The network model in Sect. 17 differs from the protocol level in that it includes additional stream objects, and transitions related to them.
- The abstraction function in Sect. 18 ties the protocol-level and the service-level specifications together.

The rest of this document consists of the HOL specification itself. This 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 the specification one should first look at the key types used (base types from the protocol level, the service-level host types, and the stream types) and then browse the Host LTS Socket Call rules.

The service-level and the protocol-level specifications share common theory infrastructure: the service-level specification imports all protocol-level theories upto and including `TCP1_preHostTypes`. These theories are not duplicated here; they can be found in the protocol-level specification [7].

1.8 Project History

In this section we summarise our previous work that led up to this TCP specification, to put it in context. All of these, and the HOL theories for the main specifications, are available on-line[1].

Our early work focussed just on UDP, ICMP, and the Sockets API. The first technical report and TACS paper describe a model without time, threads, or modules, and using informal mathematics. The ESOP paper reports on a HOL version of the specification, extended to cover those three aspects. The SIGOPS EW paper is a position paper reflecting on the experience of this and of Norrish's C semantics work.

- The UDP Calculus: Rigorous Semantics for Real Networking. Technical report 515. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. iv+70pp. July 2001.
- The UDP Calculus: Rigorous Semantics for Real Networking. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. In TACS 2001, LNCS 2215, 535–559.
- Timing UDP: mechanized semantics for sockets, threads and failures. Keith Wansbrough, Michael Norrish, Peter Sewell, Andrei Serjantov. In ESOP 2002, LNCS 2305, 278–294.
- Rigour is good for you and feasible: reflections on formal treatments of C and UDP sockets. Michael Norrish, Peter Sewell, Keith Wansbrough. In SIGOPS EW 2002, 49–53.

The following demonstrates the feasibility of completely formal reasoning (in the Isabelle proof assistant) about executable code in a fragment of OCaml above the UDP specification:

- Stenning's Protocol Implemented in UDP and Verified in Isabelle. Michael Compton. The Australasian Theory Symposium, Jan 2005.

The next phase of the project addressed TCP and the Sockets API (including also UDP and aspects of ICMP), producing a protocol-level specification. The main specification is given in the following technical reports:

- TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, Keith Wansbrough. 88pp. Technical Report 624. March 2005.
- 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. 386pp. Technical Report 625. March 2005.

These were accompanied by papers giving a systems-oriented introduction to the work and a theory-oriented description of the specification idioms and symbolic model-checking technology used:

- Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, Keith Wansbrough. 12pp. In SIGCOMM 2005.
- Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations. Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, Keith Wansbrough. 14pp. In POPL 2006.

We then used similar techniques, but at design-time instead of after the fact, for specification and validation work on a MAC protocol for the SWIFT experimental optically switched network:

- Rigorous Protocol Design in Practice: An Optical Packet-Switch MAC in HOL. Adam Biltcliffe, Michael Dales, Sam Jansen, Tom Ridge, Peter Sewell. In ICNP 2006.
- SWIFT MAC Protocol: HOL Specification.

Returning to verification above network specifications, we demonstrated that an operational semantics network model (abstracting from our detailed service-level specification) could be integrated with a

[1] http://www.cl.cam.ac.uk/~pes20/Netsem/index.html

programming language semantics, and used for functional correctness verification of a fault-tolerant persistent distributed message queue algorithm:

- Verifying distributed systems: the operational approach. Tom Ridge. In POPL 2009.

Finally, we have the specification of this technical report, a high-level service specification related to the earlier protocol-level specification by a validated abstraction function. This introduction is an extended version of the paper:

- A rigorous approach to networking: TCP, from implementation to protocol to service. Tom Ridge, Michael Norrish, Peter Sewell. In FM'08.

1.9 Conclusion

Summary We presented a formal, mechanized, service-level specification of TCP, tackling the full detail of the real-world protocol. The specification is appropriate for formal and informal reasoning about applications built above the Sockets layer, and about the service that TCP and TCP-like protocols provide to the Sockets layer. The service-level specification stands as a precise statement of end-to-end correctness for TCP. We also presented a formal abstraction function from our previous protocol-level model of TCP to the service-level specification, thereby explaining how stream-like behaviour arises from the protocol level. We used novel validation tools, coupled with the results of previous work, to validate both the service specification and the abstraction function. The specification, abstraction function, and testing infrastructure were developed entirely in HOL.

On the practice of protocol design This service-level specification is the latest in a line of work developing rigorous techniques for real-world protocol modelling and specification [25, 28, 21, 5, 8, 4]. In most of this work to date we have focused on post-hoc specification of existing infrastructure (TCP, UDP, ICMP, and the Sockets API) rather than new protocol design, though the latter is our main goal. This is for two reasons. Firstly, the existing infrastructure is ubiquitous, and likely to remain so for the foreseeable future: these wire protocols and the Sockets API are stable articulation points around which other software shifts. It is therefore well worth characterising exactly what they are, for the benefit of both users and implementers. Secondly, and more importantly, they are excellent test cases. There has been a great deal of theoretical work on idealised protocols, but, to develop rigorous techniques that can usefully be applied, they must be tested with realistic protocols. If we can deal with TCP and Sockets, with all their accumulated legacy of corner cases and behavioural quirks, then our techniques should certainly be applicable to new protocols. We believe that that is now demonstrated, and it is confirmed by our experience with design-time formalisation and conformance testing for an experimental MAC protocol for an optically switched network [4].

In recent years there has been considerable interest in 'clean slate' networking design, and in initiatives such as FIND and GENI. Protocols developed in such work should, we argue, be developed as trios of running implementation, rigorous specification, and verified conformance tester between the two. Modest attention paid to this at design time would greatly ease the task — for example, specifying appropriate debug trace information, and carefully identifying the deterministic parts of a protocol specification, would remove the need for backtracking search during validation. Declarative specification of the intended protocol behaviour, free from the imperative control-flow imposed by typical implementation languages, enables one to see unnecessary behavioural complexities clearly. Verified conformance testing makes it possible to keep implementations and specifications in sync as they are developed. Together, they should lead to cleaner, better-understood and more robust protocols, and hence to less costly and more robust infrastructure.

More specifically to TCP, we see two main directions for future work. One is simply to scale up our validation process, covering a wide variety of common protocol stacks, increasing confidence still further by testing against more traces, identifying and testing additional invariants of connection states, and so forth, and producing a packaged conformance tester for TCP implementations. This would be useful, and on an industrial scale would be a relatively small project (compared, perhaps, to the QA effort involved in developing a new protocol stack), but doing this for an existing protocol may be inappropriate for a small research group. The weight of legacy complexity here is very large, so non-trivial resources (perhaps several man-years) would be needed to cope with the detail, but the basic scientific questions, of *how* to do this, have now been solved. Doing this for *new* protocols, on the other hand, seems clearly worthwhile, even with very limited resources.

The second, more research-oriented, question, is to consider not just validation of end-to-end functional correctness (as we have done here), but properties such as end-to-end performance. Ultimately one could envisage proving network-wide properties, such as network stability, thereby connecting highly abstract properties of these protocols to the low-level details of their implementations.

Acknowledgements We gratefully acknowledge the use of the Condor facility in the Computer Laboratory, work of Adam Biltcliffe on testing infrastructure, and support from a Royal Society University Research Fellowship (Sewell) and EPSRC grants EPC510712 and GRT11715. NICTA is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

Rule version:

Part II

TCP3_hostTypes

Chapter 2

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.

2.1 The TCP control block (TCP only)

2.1.1 Summary

tcpcb	the TCP control block

2.1.2 Rules

– **the TCP control block :**

```
tcpcb =⦇

      (* timers *)
      tt_keep : () timed option; (* keepalive timer *)

      (* other *)
      t_softerror : error option (* current transient error; reported only if failure becomes permanent *)
      (* could cut this down to the actually-possible errors? *)
    ⦈
```

2.2 Sockets (TCP and UDP)

2.2.1 Summary

tcp_socket	details of a TCP socket
protocol_info	protocol-specific socket data
socket	details of a socket
TCP_Sock0	helper constructor
TCP_Sock	helper constructor

UDP_Sock0	helper constructor
UDP_Sock	helper constructor
Sock	helper constructor
tcp_sock_of	helper accessor (beware ARBitrary behaviour on non-TCP socket)
udp_sock_of	helper accessor (beware ARBitrary behaviour on non-UDP socket)
proto_of	helper accessor
proto_eq	compare protocol of two protocol info structures

2.2.2 Rules

– **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* *)
⦈

– **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 *)
is_1 : *ip* option; (* local IP address if any *)
ps_1 : *port* option; (* local port if any *)
is_2 : *ip* option; (* remote IP address if any *)
ps_2 : *port* option; (* remote port if any *)
es : *error* option; (* pending error if any *)
cantsndmore : bool; (* output stream ends at end of send queue *)
cantrcvmore : bool; (* input stream ends at end of receive queue *)
pr : protocol_info (* protocol-specific information *)
⦈

– **helper constructor :**
TCP_Sock0(*st*, *cb*, *lis*)
=⦇ *st* := *st*; *cb* := *cb*; *lis* := *lis*⦈

– **helper constructor :**
TCP_Sock *v* = TCP_PROTO(TCP_Sock0 *v*)

– **helper constructor :**
(UDP_Sock0 : *dgram* list → *udp_socket*)*rcvq* =⦇ *rcvq* := *rcvq*⦈

– **helper constructor :**
UDP_Sock *v* = UDP_PROTO(UDP_Sock0 *v*)

– **helper constructor :**
Sock(*fid*, *sf*, is_1, ps_1, is_2, ps_2, *es*, *csm*, *crm*, *pr*)
=⦇ *fid* := *fid*; *sf* := *sf*; $is_1 := is_1$; $ps_1 := ps_1$; $is_2 := is_2$; $ps_2 := ps_2$;
es := *es*; *cantsndmore* := *csm*; *cantrcvmore* := *crm*; *pr* := *pr*⦈

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

$\text{tcp\_sock\_of}\ \ sock = \textbf{case}\ \ sock.pr\ \ \textbf{of}\ \ \text{TCP\_PROTO}(tcp\_sock) \rightarrow tcp\_sock \parallel \_ \rightarrow ARB$

– **helper accessor (beware ARBitrary behaviour on non-UDP socket) :**
$\text{udp\_sock\_of}\ \ sock = \textbf{case}\ \ sock.pr\ \ \textbf{of}\ \ \text{UDP\_PROTO}(udp\_sock) \rightarrow udp\_sock \parallel \_ \rightarrow ARB$

– **helper accessor :**
$\text{proto\_of}(\text{TCP\_PROTO}(\_1)) = PROTO\_TCP\ \wedge$
$\text{proto\_of}(\text{UDP\_PROTO}(\_3)) = PROTO\_UDP$

– **compare protocol of two protocol info structures :**
$\text{proto\_eq}\ \ pr\ pr' = (\text{proto\_of}\ \ pr = \text{proto\_of}\ \ pr')$

Description Various convenience functions.

2.3 The host (TCP and UDP)

2.3.1 Summary

host	host details
privileged_ports	
ephemeral_ports	

2.3.2 Rules

– **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) *)
params : *hostParams*(* configuration info*)
⟧

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

– **:**
$\text{privileged\_ports}\ \ h = \{Port\ n \mid n < 1024\}$

– **:**
$\text{ephemeral\_ports}\ \ h = \{Port\ n \mid n \geq h.params.min\_eph\_port \wedge n \leq h.params.max\_eph\_port\}$

Rule version: $ Id: TCP3_hostTypesScript.sml,v 1.9 2009/02/20 10:35:33 tjr22 Exp $

Description Ports below 1024 (on all systems that we know of) are reserved, and can be bound by privileged users only. Additionally there is a range of ports (1024 through 2048, 3072 or 4999 or 32768 through 61000 inclusive, depending on configuration, are used for autobinding, when no specific port is specified; these ports are called "ephemeral".

2.4 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.

2.4.1 Summary

type_abbrev_tracerecord	
tracecb_eq	compare two control blocks for "equality" modulo known issues
tracesock_eq	compare two sockets for "equality" modulo known issues

2.4.2 Rules

– :
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)
= **T** (* placeholder *)

– 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 is_2, ps_2 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) *)

$$
\begin{array}{l}
(\textbf{case}\ \ quad\ \ \textbf{of} \\
\quad \uparrow(is_1, ps_1, is_2, ps_2) \rightarrow is_1 = sock.is_1 \wedge \\
\qquad\qquad ps_1 = sock.ps_1 \wedge \\
\qquad\qquad (\textbf{if}\ \ flav = TA\_DROP\ \ \textbf{then}\ \ \textbf{T}\ \ \textbf{else}\ \ is_2 = sock.is_2) \wedge \\
\qquad\qquad (\textbf{if}\ \ flav = TA\_DROP\ \ \textbf{then}\ \ \textbf{T}\ \ \textbf{else}\ \ ps_2 = sock.ps_2) \parallel \\
\quad * \rightarrow \textbf{T}) \wedge \\
st = tcp\_sock.st \wedge \\
\text{tracecb\_eq}\ flav\ st\ sock.es\ cb\ tcp\_sock.cb)
\end{array}
$$

Part III

TCP3_streamTypes

Chapter 3

Stream types

This file defines types for streams: stream control information to represent control messages on the wire, a unidirectional stream, and a bidirectional stream.

3.1 Stream types (TCP and UDP)

Basic stream types.

3.1.1 Summary

type_abbrev_streamid	
streamFlags	stream control information
tcpStream	unidirectional stream
tcpStreams	bidirectional stream

3.1.2 Rules

– **:**
type_abbrev *streamid* : (*ip*#*port*)set

– **stream control information :**
streamFlags =⟨[
SYN : bool; (* *SYN*, no *ACK* *)
SYNACK : bool; (* *SYN* with *ACK* *)
FIN : bool;
RST : bool
]⟩

Description
We record stream control-flow information with each unidirectional stream. This corresponds to the protocol-level control-flow messages. For example, the *SYNACK* flag is set iff there is a message at the protocol-level (in queues or on the wire) that has both the *SYN* and the *ACK* flags set, and which may be received as a valid message. A message may not be valid if, for example, the sequence number is out of order.

– **unidirectional stream :**

tcpStream =⟦
i : *ip*; (* source IP *)
p : *port*; (* source port *)
flgs : streamFlags;
data : byte list;
destroyed : bool
⟧

Description

The *ip* and *port* record the origin of the stream, which is primarily used to identify a unidirectional stream in an unordered pair of streams. The *flgs* record the control-flow information. The *data* is simply a list of bytes. The *destroyed* flag records whether the socket has been closed at the source, or perhaps removed altogether.

– **bidirectional stream :**
tcpStreams =⟦ *streams* : tcpStream set⟧

Description

A bidirectional stream is an unordered pair of streams, thus, we expect that there are always two tcpStreams in *streams*.

Part IV

TCP3_host0

Chapter 4

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.

4.1 Transition labels (TCP and UDP)

Host transition labels.

4.1.1 Summary

Lhost0	Host transition labels

4.1.2 Rules

– **Host transition labels :**
Lhost0 =
(* library interface *)
Lh_call **of** *tid*#*LIB_interface* (* invocation of LIB call, written e.g. $tid \cdot (\mathsf{socket}(socktype))$ *)
| Lh_return **of** *tid*#*TLang* (* return result of LIB call, written $\overline{tid \cdot v}$ *)

(* message transmission and receipt *)
| Lh_senddatagram **of** *msg* (* output of message to the network, written $\overline{msg}$ *)
| Lh_recvdatagram **of** *msg* (* input of message from the network, written *msg* *)
| Lh_loopdatagram **of** *msg* (* loopback output/input, written $\overleftrightarrow{msg}$ *)

(* connectivity changes *)
| Lh_interface **of** *ifid*#bool (* set interface status to boolean *up*, written Lh_interface(*ifid*, *up*) *)

(* miscellaneous *)
| τ (* internal transition, written τ *)
| Lh_trace **of** *tracerecord* (* TCP trace record, written Lh_trace *tr* *)

Rule version: $ Id: TCP3_host0Script.sml,v 1.5 2009/02/19 17:47:27 tjr22 Exp $

Part V

TCP3_auxFns

Chapter 5

Auxiliary functions

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

5.1 Stream versions of routing functions (TCP and UDP)

5.1.1 Summary

stream_test_outroute	if destination IP specified, do *test_outroute_ip*
stream_loopback_on_wire	check if a message bears a loopback address

5.1.2 Rules

– **if destination IP specified, do** *test_outroute_ip* **:**

$$\mathit{stream\_test\_outroute}(is_2, \mathit{rttab}, \mathit{ifds}, \mathit{arch})$$
$$= \textbf{case}\ \ is_2\ \ \textbf{of}$$
$$\quad \uparrow i_2 \rightarrow \uparrow(\mathit{test\_outroute\_ip}(i_2, \mathit{rttab}, \mathit{ifds}, \mathit{arch}))$$
$$\| \ \_ \rightarrow *$$

Description Version for streams.

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

$$\mathit{stream\_loopback\_on\_wire}(is_1, is_2)(\mathit{ifds} : \mathit{ifid} \mapsto \mathit{ifd}) =$$
$$\textbf{case}\ (is_1, is_2)\ \textbf{of}$$
$$\quad (*, *) \rightarrow \mathbf{F}$$
$$\| \ (*, \uparrow j) \rightarrow \mathbf{F}$$
$$\| \ (\uparrow i, *) \rightarrow \mathbf{F}$$
$$\| \ (\uparrow i, \uparrow j) \rightarrow \mathit{in\_loopback}\ i \land \neg \mathit{in\_local}\ \mathit{ifds}\ j$$

Description Version for streams.

5.2 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 *fid*s. This table is modelled by a finite map. File descriptors are isomorphic to the natural numbers.

5.2.1 Summary

sane_socket	socket sanity invariants hold

5.2.2 Rules

– **socket sanity invariants hold :**
sane_socket $sock = \mathbf{T}$

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.

5.3 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.

5.3.1 Summary

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

5.3.2 Rules

– **the set of ports currently bound by a socket for a protocol :**
bound_ports_protocol_autobind $pr\ socks = \{p \mid \exists s : socket.$
$s\ \in \mathbf{rng}(socks) \wedge s.ps_1 = \uparrow p\ \wedge$
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 $pr\ socks\ sf\ arch\ is\ p =$
$p\ \notin$
$\{port \mid \exists s : socket.$
$s\ \in \mathbf{rng}(socks) \land s.ps_1 = \uparrow port \land$
proto_eq $s.pr\ pr \land$
(**if** $bsd\_arch\ arch \land SO\_REUSEADDR\ \in\ sf.b$ **then**
$s.is_2 = * \land s.is_1 = is$
else if $linux\_arch\ arch \land SO\_REUSEADDR\ \in\ sf.b \land SO\_REUSEADDR\ \in\ s.sf.b \land$
$((\exists tcp\_sock.\mathrm{TCP\_PROTO}(tcp\_sock) = s.pr \land \neg(tcp\_sock.st = LISTEN)) \lor$
$\exists udp\_sock.\mathrm{UDP\_PROTO}(udp\_sock) = s.pr)$ **then**
F(* If socket is not in LISTEN state or is a UDP socket can always rebind here *)
else if $windows\_arch\ arch \land SO\_REUSEADDR\ \in\ sf.b$ **then**
F(* can rebind any UDP address; not sure about TCP - assume the same for now *)
else
$(is = * \lor s.is_1 = * \lor (\exists i : ip.is = \uparrow i \land s.is_1 = \uparrow i)))\}$

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$(\uparrow p, _, _, _) = \{p\} \land$
autobind$(*, pr, h, socks) =$ (ephemeral_ports h) **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 *) $\land$
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$(_, *, _, _)_ = 0n) \land$
(match_score$(*, \uparrow p_1, *, *)(i_3, ps3, i_4, ps4) =$
if $ps4 = \uparrow p_1$ **then** 1 **else** 0) $\land$
(match_score$(\uparrow i_1, \uparrow p_1, *, *)(i_3, ps3, i_4, ps4) =$
if $(i_1 = i_4) \land (\uparrow p_1 = ps4)$ **then** 2 **else** 0) $\land$
(match_score$(\uparrow i_1, \uparrow p_1, \uparrow i_2, *)(i_3, ps3, i_4, ps4) =$
if $(i_2 = i_3) \land (i_1 = i_4) \land (\uparrow p_1 = ps4)$ **then** 3 **else** 0) $\land$
(match_score$(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)(i_3, ps3, i_4, ps4) =$
if $(\uparrow p_2 = ps3) \land (i_2 = i_3) \land (i_1 = i_4) \land (\uparrow p_1 = ps4)$ **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 agrument) 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 :**

$$
\begin{array}{l}
\text{lookup\_udp}\ \mathit{socks}\ \mathit{quad}\ \mathit{bound}\ \mathit{arch} = \\
\quad \{\mathit{sid} \mid \mathit{sid} \in \mathbf{dom}(\mathit{socks}) \wedge \\
\qquad \mathbf{let}\ s = \mathit{socks}[\mathit{sid}]\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathit{sn} = \text{match\_score}(s.\mathit{is}_1, s.\mathit{ps}_1, s.\mathit{is}_2, s.\mathit{ps}_2)\mathit{quad}\ \mathbf{in} \\
\qquad\quad \mathit{sn} > 0 \wedge \\
\qquad\quad \mathbf{if}\ \mathit{windows\_arch}\ \mathit{arch}\ \mathbf{then} \\
\qquad\qquad \mathbf{if}\ \mathit{sn} = 1\ \mathbf{then} \\
\qquad\qquad\quad \neg(\exists(\mathit{sid}', s') :: (\mathit{socks} \backslash\backslash \mathit{sid}).\ \text{match\_score}(s'.\mathit{is}_1, s'.\mathit{ps}_1, s'.\mathit{is}_2, s'.\mathit{ps}_2)\mathit{quad} > \mathit{sn}) \\
\qquad\qquad \mathbf{else}\ \mathbf{T} \\
\qquad\quad \mathbf{else} \\
\qquad\qquad \neg(\exists(\mathit{sid}', s') :: (\mathit{socks} \backslash\backslash \mathit{sid}). \\
\qquad\qquad\qquad (\text{match\_score}(s'.\mathit{is}_1, s'.\mathit{ps}_1, s'.\mathit{is}_2, s'.\mathit{ps}_2)\mathit{quad} > \mathit{sn} \vee \\
\qquad\qquad\qquad (\mathit{linux\_arch}\ \mathit{arch} \wedge \text{match\_score}(s'.\mathit{is}_1, s'.\mathit{ps}_1, s'.\mathit{is}_2, s'.\mathit{ps}_2)\mathit{quad} = \mathit{sn} \wedge \\
\qquad\qquad\qquad\quad \text{bound\_after}\ \mathit{sid}'\ \mathit{sid}\ \mathit{bound})))\}
\end{array}
$$

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.

Variations

FreeBSD	For FreeBSD, the set contains the sockets for which the score is greater than zero and there is no other socket in *socks* 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, $sn = 1$, and there are no sockets with greater scores.

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

$$
\begin{array}{l}
\text{tcp\_socket\_best\_match}(\mathit{socks} : \mathit{sid} \mapsto \mathsf{socket})(\mathit{sid}, \mathit{sock})(\mathit{seg} : \mathit{tcpSegment})\mathit{arch} = \\
\text{(* is the socket sid the best match for segment seg? *)} \\
\mathbf{let}\ s = \mathit{sock}\ \mathbf{in} \\
\mathbf{let}\ \mathit{score} = \text{match\_score}(s.\mathit{is}_1, s.\mathit{ps}_1, s.\mathit{is}_2, s.\mathit{ps}_2) \\
\qquad\qquad (\mathbf{the}\ \mathit{seg}.\mathit{is}_1, \mathit{seg}.\mathit{ps}_1, \mathbf{the}\ \mathit{seg}.\mathit{is}_2, \mathit{seg}.\mathit{ps}_2)\ \mathbf{in} \\
\neg(\exists(\mathit{sid}', s') :: \mathit{socks} \backslash\backslash \mathit{sid}. \\
\qquad \text{match\_score}(s'.\mathit{is}_1, s'.\mathit{ps}_1, s'.\mathit{is}_2, s'.\mathit{ps}_2) \\
\qquad\qquad (\mathbf{the}\ \mathit{seg}.\mathit{is}_1, \mathit{seg}.\mathit{ps}_1, \mathbf{the}\ \mathit{seg}.\mathit{is}_2, \mathit{seg}.\mathit{ps}_2) > \mathit{score})
\end{array}
$$

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) 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.

Rule version: $ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp $

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

$$
\begin{array}{l}
\text{lookup\_icmp}\ \mathit{socks}\ \mathit{icmp}\ \mathit{arch}\ \mathit{bound} = \\
\quad \{\mathit{sid0} \mid \exists(\mathit{sid},\mathit{sock}) :: \mathit{socks}. \\
\qquad\qquad \mathit{sock.ps}_1 = \mathit{icmp.ps3} \wedge \text{proto\_of}\ \mathit{sock.pr} = \mathit{icmp.proto} \wedge \mathit{sid0} = \mathit{sid} \wedge \\
\qquad\qquad \textbf{if}\ \mathit{windows\_arch}\ \mathit{arch}\ \textbf{then}\ \mathbf{T} \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad \mathit{sock.is}_1 = \mathit{icmp.is3} \wedge \mathit{sock.is}_2 = \mathit{icmp.is4} \wedge \\
\qquad\qquad\quad (\mathit{sock.ps}_2 = \mathit{icmp.ps4} \vee \\
\qquad\qquad\quad (\mathit{linux\_arch}\ \mathit{arch} \wedge \\
\qquad\qquad\qquad \text{proto\_of}\ \mathit{sock.pr} = \mathit{PROTO\_UDP} \wedge \mathit{sock.ps}_2 = * \wedge \\
\qquad\qquad\qquad \neg(\exists(\mathit{sid}', s) :: (\mathit{socks} \backslash\backslash \mathit{sid}). \\
\qquad\qquad\qquad\qquad s.\mathit{is}_1 = \mathit{icmp.is3} \wedge s.\mathit{is}_2 = \mathit{icmp.is4} \wedge \\
\qquad\qquad\qquad\qquad s.\mathit{ps}_1 = \mathit{icmp.ps3} \wedge s.\mathit{ps}_2 = \mathit{icmp.ps4} \wedge \\
\qquad\qquad\qquad\qquad \text{proto\_of}\ s.\mathit{pr} = \mathit{icmp.proto} \wedge \\
\qquad\qquad\qquad\qquad \text{bound\_after}\ \mathit{sid}'\ \mathit{sid}\ \mathit{bound}) \\
\qquad\qquad\quad))\}
\end{array}
$$

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.

5.4 TCP Options (TCP only)

TCP option handling.

5.4.1 Summary

do_tcp_options	Constrain the TCP timestamp option values that appear in an outgoing segment
calculate_tcp_options_len	Calculate the length consumed by the TCP options in a real TCP segment

5.4.2 Rules

Rule version: $ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp $

– **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* 0*w*) **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.

5.5 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.

5.5.1 Summary

calculate_buf_sizes	Calculate buffer sizes for *rcvbufsize*, *sndbufsize*, *t_maxseg*, and *snd_cwnd*
send_queue_space	

5.5.2 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_tstmp 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_tstmp*)
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 *rounddown 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**

let $do\_rfc3390 =$ **T** **in**

(* compute initial cwnd *)
let $snd\_cwnd =$
if $do\_rfc3390$ **then** **min**$(4 * t\_maxseg''')$(**max**$(2 * t\_maxseg''')4380)$
else
$(t\_maxseg''' *$ (**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*.

– :
send_queue_space($sndq\_max$: num)$sndq\_size\ oob\ arch\ maxseg\ i_2 =$
$\{n \mid$ **if** $bsd\_arch\ arch$ **then**
$n \leq (sndq\_max - sndq\_size) +$ (**if** oob **then** oob_extra_sndbuf **else** 0)
else if $linux\_arch\ arch$ **then**
(**if** $in\_loopback\ i_2$ **then**
$n = maxseg + ((sndq\_max - sndq\_size)$ **div** $16816) * maxseg$
else
$n = (2 * maxseg) + ((sndq\_max - sndq\_size - 1890)$ **div** $1888) * maxseg$)
else $n \geq 0\}$

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.

5.6 UDP support (UDP only)

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

5.6.1 Summary

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

5.6.2 Rules

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

$$
\begin{aligned}
&(\mathrm{dosend}(\mathit{ifds}, \mathit{rttab}, (*, \mathit{data}), (\uparrow i_1, \uparrow p_1, \uparrow i_2, \mathit{ps}_2), \mathit{oq}, \mathit{oq}', \mathit{ok}) = \\
&\mathit{enqueue\_oq}(\mathit{oq}, \mathit{UDP}(\langle\!|\; \mathit{is}_1 := \uparrow i_1; \mathit{is}_2 := \uparrow i_2; \\
&\qquad \mathit{ps}_1 := \uparrow p_1; \mathit{ps}_2 := \mathit{ps}_2; \\
&\qquad \mathit{data} := \mathit{data} |\!\rangle), \\
&\quad \mathit{oq}', \mathit{ok})) \wedge \\
&(\mathrm{dosend}(\mathit{ifds}, \mathit{rttab}, (\uparrow(i, p), \mathit{data}), (*, \uparrow p_1, *, *), \mathit{oq}, \mathit{oq}', \mathit{ok}) = \\
&(\exists i_1'.\mathit{enqueue\_oq}(\mathit{oq}, \mathit{UDP}(\langle\!|\; \mathit{is}_1 := \uparrow i_1'; \mathit{is}_2 := \uparrow i; \\
&\qquad \mathit{ps}_1 := \uparrow p_1; \mathit{ps}_2 := \uparrow p; \\
&\qquad \mathit{data} := \mathit{data} |\!\rangle), \\
&\quad \mathit{oq}', \mathit{ok}) \wedge i_1' \in \mathit{auto\_outroute}(i, *, \mathit{rttab}, \mathit{ifds}))) \wedge \\
&(\mathrm{dosend}(\mathit{ifds}, \mathit{rttab}, (\uparrow(i, p), \mathit{data}), (\uparrow i_1, \uparrow p_1, \mathit{is}_2, \mathit{ps}_2), \mathit{oq}, \mathit{oq}', \mathit{ok}) = \\
&\mathit{enqueue\_oq}(\mathit{oq}, \mathit{UDP}(\langle\!|\; \mathit{is}_1 := \uparrow i_1; \mathit{is}_2 := \uparrow i; \\
&\qquad \mathit{ps}_1 := \uparrow p_1; \mathit{ps}_2 := \uparrow p; \\
&\qquad \mathit{data} := \mathit{data} |\!\rangle), \\
&\quad \mathit{oq}', \mathit{ok}))
\end{aligned}
$$

Description For use in UDP *sendto*().

5.7 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.

5.7.1 Summary

next_smaller	find next-smaller element of a set
mtu_tab	path MTU plateaus to try

5.7.2 Rules

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

$$(\mathrm{next\_smaller} : (\mathsf{num} \rightarrow \mathsf{bool}) \rightarrow \mathsf{num} \rightarrow \mathsf{num})\mathit{xs}\ y = @x :: \mathit{xs}.x < y \wedge \forall x' :: \mathit{xs}.x' > x \implies x' \geq y$$

– **path MTU plateaus to try :**

$$
\begin{aligned}
&\mathrm{mtu\_tab}\ \mathit{arch} = \mathbf{if}\ \ \mathit{linux\_arch}\ \mathit{arch}\ \ \mathbf{then} \\
&\qquad \{32000; 17914; 8166; 4352; 2002; 1492; 576; 296; 216; 128; 68\} : \mathsf{num\ set} \\
&\quad \mathbf{else} \\
&\qquad \{65535; 32000; 17914; 8166; 4352; 2002; 1492; 1006; 508; 296; 68\}
\end{aligned}
$$

Rule version: $ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp $

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.

5.8 The initial TCP control block (TCP only)

The initial state of the TCP control block.

5.8.1 Summary

initial_cb

5.8.2 Rules

– **:**
initial_cb =
⦇
  tt_keep := ∗;
  t_softerror := ∗
⦈

Chapter 6

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.

6.1 General Segment Creation (TCP only)

The TCP output routines. These, together with the input routines in *deliver_in_3*, form the heart of TCP.

6.1.1 Summary

tcp_output_required	determine whether TCP output is required
tcp_output_really	do TCP output
stream_tcp_output_really	do TCP output
tcp_output_perhaps	combination of tcp_output_required and tcp_output_really
stream_tcp_output_perhaps	combination of tcp_output_required and tcp_output_really

6.1.2 Rules

– determine whether TCP output is required :
tcp_output_required($do\_output, persist\_fun$) =
($do\_output \ \in \{\mathbf{T}; \mathbf{F}\} \wedge$
$persist\_fun \ \in \{*; \uparrow(\lambda cb : tcpcb.cb)\}$)

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 $sock(sock', outsegs')$ =
let tcp_sock = tcp_sock_of $sock$ **in**

let $cb = tcp\_sock.cb$ **in**

(* Assert that the socket is fully bound and connected *)
$sock.is_1 \neq * \wedge$
$sock.is_2 \neq * \wedge$
$sock.ps_1 \neq * \wedge$
$sock.ps_2 \neq * \wedge$

(* Is it possible that a *FIN* may need to be transmitted? *)
let $fin\_required = (sock.cantsndmore \wedge tcp\_sock.st \notin \{FIN\_WAIT\_2; TIME\_WAIT\})$ **in**

(* Should *FIN* be set in this segment? *)
choose $snd\_nxt\_plus\_length\_data\_to\_send\_ge\_last\_sndq\_data\_seq :: \{\mathbf{T}; \mathbf{F}\}$.
let $FIN = (fin\_required \wedge snd\_nxt\_plus\_length\_data\_to\_send\_ge\_last\_sndq\_data\_seq)$ **in**

$\exists snd\_nxt'\ rcv\_nxt\ URG\ ACK\ PSH\ win\ urp\_\ ts\ data\_to\_send.$
let $seg = \langle\!| is_1 := sock.is_1;$
$is_2 := sock.is_2;$
$ps_1 := sock.ps_1;$
$ps_2 := sock.ps_2;$
$seq := snd\_nxt';$
$ack := rcv\_nxt;$
$URG := URG;$
$ACK := ACK;$
$PSH := PSH;$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win;$
$ws := *;$
$urp := urp\_;$
$mss := *;$
$ts := ts;$
$data := data\_to\_send$
$|\!\rangle$ **in**

(* If emitting a *FIN* for the first time then change TCP state *)
let $st' =$ **if** FIN **then**
case $tcp\_sock.st$ **of**
$SYN\_SENT \rightarrow tcp\_sock.st \parallel$ (* can't move yet – wait until connection established (see *deliver_in_2*/*deliver_in_3*) *)
$SYN\_RECEIVED \rightarrow tcp\_sock.st \parallel$ (* can't move yet – wait until connection established (see *deliver_in_2*/*deliver_in_3*) *)
$ESTABLISHED \rightarrow FIN\_WAIT\_1 \parallel$
$CLOSE\_WAIT \rightarrow LAST\_ACK \parallel$
$FIN\_WAIT\_1 \rightarrow tcp\_sock.st \parallel$ (* FIN retransmission *)
$FIN\_WAIT\_2 \rightarrow tcp\_sock.st \parallel$ (* can't happen *)
$CLOSING \rightarrow tcp\_sock.st \parallel$ (* FIN retransmission *)
$LAST\_ACK \rightarrow tcp\_sock.st \parallel$ (* FIN retransmission *)
$TIME\_WAIT \rightarrow tcp\_sock.st$ (* can't happen *)
else
$tcp\_sock.st$ **in**

(* Update the socket *)
$sock' = sock \langle\!| pr := \mathrm{TCP\_PROTO}(tcp\_sock \langle\!| st := st' |\!\rangle) |\!\rangle \wedge$

(* 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.

– **do TCP output :**

stream_tcp_output_really sock$(sock', FIN)$ =
let *tcp_sock* = tcp_sock_of *sock* **in**
let *cb* = *tcp_sock.cb* **in**

(* Assert that the socket is fully bound and connected *)
$sock.is_1 \neq * \wedge$
$sock.is_2 \neq * \wedge$
$sock.ps_1 \neq * \wedge$
$sock.ps_2 \neq * \wedge$

(* 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**

(* Should *FIN* be set in this segment? *)
choose *snd_nxt_plus_length_data_to_send_ge_last_sndq_data_seq* :: {**T**; **F**}.
FIN = (*fin_required* ∧ *snd_nxt_plus_length_data_to_send_ge_last_sndq_data_seq*) ∧

(* 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**

(* Update the socket *)
sock' = *sock* ⦃ *pr* := TCP_PROTO(*tcp_sock* ⦃ *st* := *st'* ⦄) ⦄

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 $sock(sock', outsegs) =$
$\exists do\_output\ persist\_fun.$
(tcp_output_required$(do_output, persist_fun) \land$
let $sock'' = sock$ **in**
if do_output **then**
tcp_output_really $sock''(sock', outsegs)$
else
$(sock' = sock'' \land outsegs = []))$

– **combination of** tcp_output_required **and** tcp_output_really **:**
(* *FINs* argument records whether any messages were sent, and if so, whether they were a *FIN* *)
$stream\_tcp\_output\_perhaps\ sock(sock', FINs) =$
$\exists do\_output\ persist\_fun.$
(tcp_output_required$(do_output, persist_fun) \land$
let $sock'' = sock$ **in**
if do_output **then**
$\exists FIN.$
$stream\_tcp\_output\_really\ sock''(sock', FIN)$(* definitely does send a seg *) $\land$
$FINs = \uparrow FIN$
else
$(sock' = sock'' \land FINs = *))$

6.2 Segment Queueing (TCP only)

Once a segment is generated for output, it must be enqueued for transmission. This enqueuing may fail. These functions model what happens in this case, and encapsulate the enqueuing-and-possibly-rolling-back process.

6.2.1 Summary

rollback_tcp_output	Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails
stream_rollback_tcp_output	Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails
enqueue_or_fail	wrap rollback_tcp_output together with enqueue
stream_enqueue_or_fail	wrap rollback_tcp_output together with enqueue
stream_enqueue_or_fail_sock	version of enqueue_or_fail that works with sockets rather than cbs
enqueue_and_ignore_fail	version of enqueue_or_fail that ignores errors and doesn't touch the tcpcb
enqueue_each_and_ignore_fail	version of above that ignores errors and doesn't touch the tcpcb
stream_mlift_tcp_output_perhaps_or_fail	do mliftc for function returning at most one segment and not dealing with queueing flag

6.2.2 Rules

– **Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails :**
rollback_tcp_output $rcvdsyn\ seg\ arch\ rttab\ ifds\ is\_connect\ cb\_in(cb', es', outsegs') =$

(* NB: from cb_0, 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** $\{\mathbf{T}\}$ **else** $\{\mathbf{T}; \mathbf{F}\})$.

Rule version: $ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp $

let *route* = *test_outroute*(*seg*, *rttab*, *ifds*, *arch*) **in**
let *f1* = λ*cb*.**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**
if ¬*allocated* **then** (* allocation failure *)
$cb' = cb\_in \wedge outsegs' = [\,] \wedge es' = \uparrow ENOBUFS$
else if *route* = ∗ **then** (* ill-formed segment *)
ASSERTION_FAILURE"rollback_tcp_output:1"(* should never happen *)
else if ∃*e*.*route* = ↑(↑ *e*) **then** (* routing failure *)
$cb' = f1\ cb\_in \wedge outsegs' = [\,] \wedge es' =$ **the** *route*
else if *loopback_on_wire seg ifds* **then** (* loopback not allowed on wire - RFC1122 *)
(**if** *windows_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = [\,] \wedge es' = *$(* Windows silently drops segment! *)
else if *bsd_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = [\,] \wedge es' = \uparrow EADDRNOTAVAIL$
else if *linux_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = [\,] \wedge es' = \uparrow EINVAL$
else
ASSERTION_FAILURE"rollback_tcp_output:2"(* never happen *)
)
else
(∃*queued*.
$outsegs' = [(seg, queued)] \wedge$
if ¬*queued* **then** (* queueing failure *)
$cb' = cb\_in \wedge es' = \uparrow ENOBUFS$
else (* success *)
$cb' = cb\_in \wedge es' = *$)
)

– **Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails :**
stream_rollback_tcp_output rcvdsyn(is_1, is_2)*arch rttab ifds cb_in*$(cb', es', outsegs') =$

(* NB: from cb_0, 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* = *stream_test_outroute*(is_2, *rttab*, *ifds*, *arch*) **in**
let *f1* = λ*cb*.**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**
if ¬*allocated* **then** (* allocation failure *)
$cb' = cb\_in \wedge outsegs' = \mathbf{F} \wedge es' = \uparrow ENOBUFS$
else if *route* = ∗ **then** (* ill-formed segment *)
ASSERTION_FAILURE"stream_rollback_tcp_output:1"(* should never happen *)
else if ∃*e*.*route* = ↑(↑ *e*) **then** (* routing failure *)
$cb' = f1\ cb\_in \wedge outsegs' = \mathbf{F} \wedge es' =$ **the** *route*
else if *stream_loopback_on_wire*(is_1, is_2)*ifds* **then** (* loopback not allowed on wire - RFC1122 *)
(**if** *windows_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = \mathbf{F} \wedge es' = *$(* Windows silently drops segment! *)
else if *bsd_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = \mathbf{F} \wedge es' = \uparrow EADDRNOTAVAIL$
else if *linux_arch arch* **then**
$cb' = cb\_in \wedge outsegs' = \mathbf{F} \wedge es' = \uparrow EINVAL$
else
ASSERTION_FAILURE"stream_rollback_tcp_output:2"(* never happen *)
)

else
$(\exists queued.$
$outsegs' = \mathbf{T} \wedge$
if $\neg queued$ **then** (* queueing failure *)
$cb' = cb\_in \wedge es' = \uparrow ENOBUFS$
else (* success *)
$cb' = cb\_in \wedge es' = *)$
)

– **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**
$[\,] \rightarrow cb' = cb_0 \wedge oq' = oq$
$\|\ [seg] \rightarrow (\exists outsegs'\ es'.$
rollback_tcp_output $rcvdsyn\ seg\ arch\ rttab\ ifds$ $\mathbf{F}$(* X cb0 X *)$cb\_in(cb', es', outsegs') \wedge$
$enqueue\_oq\_list\_qinfo(oq, outsegs', oq'))$
$\|\ \_other84 \rightarrow ASSERTION\_FAILURE$"enqueue_or_fail"(* only 0 or 1 segments at a time *)
)

– **wrap** rollback_tcp_output **together with enqueue :**
$stream\_enqueue\_or\_fail\ rcvdsyn\ arch\ rttab\ ifds(is_1, is_2)cb\_in\ cb' =$
$(\exists es'\ outsegs'.stream\_rollback\_tcp\_output\ rcvdsyn(is_1, is_2)arch\ rttab\ ifds\ cb\_in(cb', es', outsegs'))$

– **version of** enqueue_or_fail **that works with sockets rather than cbs :**
$stream\_enqueue\_or\_fail\_sock\ rcvdsyn\ arch\ rttab\ ifds(is_1, is_2)sock0\ sock\ sock' =$
(* 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**
$(\exists cb'.$
$stream\_enqueue\_or\_fail\ rcvdsyn\ arch\ rttab\ ifds(is_1, is_2)$(tcp_sock_of $sock).cb\ cb' \wedge$
$sock' = sock\ \llparenthesis\ pr :=$ TCP_PROTO(tcp_sock_of $sock\ \llparenthesis$
$cb := cb'$
$\rrparenthesis)\rrparenthesis)$

– **version of** enqueue_or_fail **that ignores errors and doesn't touch the tcpcb :**
enqueue_and_ignore_fail $arch\ rttab\ ifds\ outsegs\ oq\ oq' =$
$\exists rcvdsyn\ cb_0\ cb\_in\ cb'.$
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 $arch\ rttab\ ifds[\,]oq\ oq' = (oq = oq')) \wedge$
(enqueue_each_and_ignore_fail $arch\ rttab\ ifds(seg :: segs)oq\ oq''$
$= \exists oq'.$ enqueue_and_ignore_fail $arch\ rttab\ ifds[seg]oq\ oq' \wedge$
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 :**
$stream\_mlift\_tcp\_output\_perhaps\_or\_fail$(* X ts_val X *)$arch\ rttab\ ifds_0\ s(s', FIN) =$
$\exists s_1\ FINs.$
$stream\_tcp\_output\_perhaps\ s(s_1, FINs) \wedge$
case $FINs$ **of**
$* \rightarrow s' = s_1 \wedge FIN = \mathbf{F}$
$\|\ \uparrow FIN' \rightarrow (\exists cb'\ es'\ outsegs'.$(* ignore error return *)
$stream\_rollback\_tcp\_output$ $\mathbf{T}(s_1.is_1, s_1.is_2)arch\ rttab\ ifds_0$
(* X (tcp_sock_of s).cb X *)(tcp_sock_of $s_1).cb(cb', es', outsegs') \wedge$
$s' = s_1\ \llparenthesis\ pr :=$ TCP_PROTO(tcp_sock_of $s_1\ \llparenthesis\ cb := cb' \rrparenthesis)\rrparenthesis \wedge$
$FIN = (outsegs' \wedge FIN'))$

6.3 Incoming Segment Functions (TCP only)

Updates performed to the idle, keepalive, and `FIN_WAIT_2` timers for every incoming segment.

6.3.1 Summary

update_idle	Do updates appropriate to receiving a new segment on a connection

6.3.2 Rules

– **Do updates appropriate to receiving a new segment on a connection :**
update_idle *tcp_sock* tt_keep' =
choose *tf_needfin* :: {**T**; **F**}.
tt_keep' = (**if** $\neg$(*tcp_sock.st* = *SYN_RECEIVED* $\wedge$ *tf_needfin*) **then**
(* reset keepalive timer to 2 hours. *)
$\uparrow((())_{slow\_timer\ TCPTV\_KEEP\_IDLE})$
else
tcp_sock.cb.tt_keep)

6.4 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.

6.4.1 Summary

dropwithreset	emit a RST segment corresponding to the passed segment, unless that would be stupid.
stream_mlift_dropafterack_or_fail	send immediate ACK to segment, but otherwise process it no further

6.4.2 Rules

– **emit a RST segment corresponding to the passed segment, unless that would be stupid. :**
dropwithreset *segRST*(is_1, is_2)*ifds*$_0$ *RST* =
(* 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 *)
segRST $\vee$
(* is segment a (link-layer?) broadcast or multicast? *)
F $\vee$
(* is source or destination broadcast or multicast? *)
$(\exists i_1.is_1 = \uparrow i_1 \wedge is\_broadormulticast\ \emptyset\ i_1) \vee$
$(\exists i_2.is_2 = \uparrow i_2 \wedge is\_broadormulticast\ ifds_0\ i_2)$
(* BSD only checks incoming interface, but should have same effect as long as interfaces don't overlap *)
then
RST = **F**
else
RST $\in$ {**T**; **F**}

Rule version: $ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp $

– **send immediate ACK to segment, but otherwise process it no further :**
$stream\_mlift\_dropafterack\_or\_fail\ segRST\ arch\ rttab\ ifds\ sock(sock', FIN, RST, stop') =$
(* *ifds* is just in case we need to send a RST, to make sure we don't send it to a broadcast address. *)
let $continue = \neg stop'$ **in**
let $tcp\_sock =$ tcp_sock_of $sock$ **in**
$(continue = \mathbf{T} \wedge$
let $cb = tcp\_sock.cb$ **in**
choose $ACK :: \{\mathbf{T}; \mathbf{F}\}.$
choose $ack\_lt\_snd\_una\_or\_snd\_max\_lt\_ack :: \{\mathbf{T}; \mathbf{F}\}.$
if $tcp\_sock.st = SYN\_RECEIVED \wedge$
$ACK \wedge$
$ack\_lt\_snd\_una\_or\_snd\_max\_lt\_ack$
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`) *)
$sock' = sock \wedge$
$FIN = \mathbf{F} \wedge$
dropwithreset $segRST(sock.is_1, sock.is_2) ifds\ RST$
(* ignore queue full error *)
else
$(\exists sock_1\ msgFIN.$(* ignore errors *)
let $tcp\_sock1 =$ tcp_sock_of $sock_1$ **in**
$stream\_tcp\_output\_really\ sock(sock_1, msgFIN) \wedge$ (* 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 *)
$\exists outsegs'\ cb'\ es'.$
$stream\_rollback\_tcp\_output\ \mathbf{T}(sock.is_1, sock.is_2) arch\ rttab\ ifds\ tcp\_sock1.cb(cb', es', outsegs') \wedge$
$sock' = sock_1\ \llparenthesis\ pr := \mathrm{TCP\_PROTO}(tcp\_sock1\ \llparenthesis\ cb := cb' \rrparenthesis) \rrparenthesis \wedge$
$FIN = ($**if** $outsegs'$ **then** $msgFIN$ **else** $\mathbf{F}) \wedge$
$RST = \mathbf{F}))$

6.5 Close Functions (TCP only)

Closing a connection, updating the socket and TCP control block appropriately.

6.5.1 Summary

tcp_close	close the socket and remove the TCPCB
tcp_drop_and_close	drop TCP connection, reporting the specified error. If synchronised, send RST to peer

6.5.2 Rules

– **close the socket and remove the TCPCB :**
tcp_close $arch\ sock = sock$
$\llparenthesis\ cantrcvmore := \mathbf{T};$ (* MF doesn't believe this is correct for Linux or WinXP *)
$cantsndmore := \mathbf{T};$
$is_1 :=$ **if** $bsd\_arch\ arch$ **then** $*$ **else** $sock.is_1;$
$ps_1 :=$ **if** $bsd\_arch\ arch$ **then** $*$ **else** $sock.ps_1;$
$pr := \mathrm{TCP_PROTO}($tcp_sock_of $sock$
$\llparenthesis\ st := CLOSED;$
$cb :=$ initial_cb$\rrparenthesis)$
$\rrparenthesis$

Rule version: \$ Id: TCP3_auxFnsScript.sml,v 1.16 2009/02/20 13:08:08 tjr22 Exp \$

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 $arch\ err\ sock(sock', (oflgs, odata : char\ \mathsf{list})) =$
let tcp_sock = tcp_sock_of $sock$ **in** (
(**if** $tcp\_sock.st \notin \{CLOSED; LISTEN; SYN\_SENT\}$ **then**
$(oflgs = oflgs \,(\!|\, SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{T} |\!)\, \wedge$
$odata \in UNIV)$
else
$(oflgs = oflgs \,(\!|\, SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F} |\!)\, \wedge$
$odata = [\,])) \wedge$
let $es' =$
if $err = \uparrow ETIMEDOUT$ **then**
(**if** $tcp\_sock.cb.t\_softerror \neq *$ **then**
$tcp\_sock.cb.t\_softerror$
else
$\uparrow ETIMEDOUT)$
else if $err \neq *$ **then** err
else $sock.es$
in
$sock'$ = tcp_close $arch(sock \,(\!|\, es := es' |\!)))$

Description BSD calls this `tcp_drop`

6.6 Socket quad testing and extraction (TCP only)

Testing and extracting the quad of a connection from the socket.

6.6.1 Summary

exists_quad_of	test whether a socket quad is set
quad_of	extract the quad from the socket

6.6.2 Rules

– **test whether a socket quad is set :**
exists_quad_of$(sock : TCP3_hostTypes\ \$socket) =$
$\exists i_1\ p_1\ i_2\ p_2.(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2) = (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2)$

– **extract the quad from the socket :**
quad_of$(sock : TCP3_hostTypes\ \$socket) =$
$(\mathbf{the}\ sock.is_1, \mathbf{the}\ sock.ps_1, \mathbf{the}\ sock.is_2, \mathbf{the}\ sock.ps_2)$

Part VI

TCP3_hostLTS

Chapter 7

Host LTS: Socket Calls

7.1 *accept*() (TCP only)

$accept : fd \rightarrow fd * (ip * port)$

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.

7.1.1 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	Ths 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.

7.1.2 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*

7.1.3 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.

7.1.4 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.

7.1.5 Summary

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

accept_1	**tcp: rc**	Return new connection; either immediately or from a blocked state.
accept_2	**tcp: block**	Block waiting for connection
accept_3	**tcp: fast fail**	Fail with *EAGAIN*: no pending connections and non-blocking semantics set
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.
accept_5	**tcp: rc**	Fail with *EINVAL*: socket not in LISTEN state
accept_6	**tcp: rc**	Fail with *EMFILE*: out of file descriptors
accept_7	**udp: fast fail**	Fail with *EOPNOTSUPP* or *EINVAL*: *accept*() called on a UDP socket

7.1.6 Rules

accept_1 **tcp: rc** **Return new connection; either immediately or from a blocked state.**

$$\begin{array}{l}
(h \, \langle\!\langle ts := ts \oplus (tid \mapsto (t)_d); \\
fds := fds; \\
files := files; \\
socks := \\
socks \oplus \\
[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \\
\qquad \text{TCP_Sock}(LISTEN, cb, \uparrow lis))); \\
(sid', \textsc{Sock}(*, sf', \uparrow i_1', \uparrow p_1, \uparrow i_2, \uparrow p_2, es', cantsndmore', cantrcvmore', \\
\qquad \text{TCP_Sock}(ESTABLISHED, cb', *)))]\rangle\!\rangle, \\
SS, MM)
\end{array}$$

$$\xrightarrow{lbl} \begin{array}{l}
(h \, \langle\!\langle ts := ts \oplus (tid \mapsto \big(Ret(OK(fd', (i_2, p_2)))\big)_{sched_timer}); \\
fds := fds'; \\
files := files \oplus [(fid', \textsc{File}(FT_Socket(sid'), ff_default))]; \\
socks := \\
socks \oplus \\
[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \\
\qquad \text{TCP_Sock}(LISTEN, cb, \uparrow lis'))); \\
(sid', \textsc{Sock}(\uparrow fid', sf', \uparrow i_1', \uparrow p_1, \uparrow i_2, \uparrow p_2, es', \\
\qquad cantsndmore', cantrcvmore', \text{TCP_Sock}(ESTABLISHED, cb', *)))]\rangle\!\rangle, \\
SS, MM)
\end{array}$$

$$\left(\begin{pmatrix}
t = Run \wedge \\
lbl = tid \cdot (accept\ fd) \wedge \\
rc = fast\ succeed \wedge \\
fid = fds[fd] \wedge \\
fd \in \mathbf{dom}(fds) \wedge \\
files[fid] = \textsc{File}(FT_Socket(sid), ff)
\end{pmatrix} \vee \begin{pmatrix}
t = Accept2(sid) \wedge \\
\quad lbl = \tau \wedge \\
\quad rc = slow\ urgent\ succeed
\end{pmatrix}\right) \wedge$$

$$\begin{array}{l}
lis.q = q \mathbin{@} [sid'] \wedge \\
lis'.q = q \wedge \\
lis'.q_0 = lis.q_0 \wedge lis'.qlimit = lis.qlimit \wedge \\
(sid \neq sid') \wedge \\
es' \neq \uparrow ECONNABORTED \wedge \\
fid' \notin ((\mathbf{dom}(files)) \cup \{fid\}) \wedge \\
nextfd\ h.arch\ fds\ fd' \wedge \\
fds' = fds \oplus (fd', fid') \wedge \\
(\forall i_1. \uparrow i_1 = is_1 \implies i_1 = i_1')
\end{array}$$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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, $socks \oplus \ldots$. 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 fid', 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 $\uparrow fid'$ and sid' is removed from the head of the completed connections queue.

When the listening socket *sid* is bound to a local IP address i_1, 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*, and an entry mapping fd' to fid' 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 $tid \cdot (accept\ fd)$ transition. The thread is left in state $Ret(OK(fd', (i_2, p_2)))$ 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**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d)\rangle\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot(accept\ fd)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Accept2(sid))_{never\_timer})\rangle\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$ff.b(O\_NONBLOCK) = \mathbf{F} \wedge$
$sid \in \mathbf{dom}(h.socks) \wedge$
$(\exists sf\ is_1\ p_1\ cb\ lis\ es.$
$\quad h.socks[sid] = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \mathbf{F}, cantrcvmore,$
$\qquad\qquad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis)) \wedge$
$\quad lis.q = [\,])$

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 p_1, is not shutdown for reading and is in blocking mode: $ff.b(O\_NONBLOCK) = \mathbf{F}$. 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. is_1 could be $*$. In this case the socket is listening for connection attempts on port p_1 for all local IP addresses.

accept_3 **tcp: fast fail** **Fail with *EAGAIN*: no pending connections and non-blocking semantics set**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d)\rangle\!\rangle, SS, MM)$$

$$\xrightarrow{tid\cdot(accept\ fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EAGAIN))_{sched\_timer})|\!\rangle, SS, MM)$$

$$
\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
h.fds[fd] = fid \wedge \\
h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge \\
ff.b(O\_NONBLOCK) = \mathbf{T} \wedge \\
sid \in \mathbf{dom}(h.socks) \wedge \\
(\exists sf\ is_1\ p_1\ cb\ lis\ es. \\
h.socks[sid] = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \\
\qquad\qquad\qquad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis)) \wedge \\
lis.q = [\,])
\end{array}
$$

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 p_1, not shutdown for writing, and in non-blocking mode: $ff.b(O\_NONBLOCK) = \mathbf{T}$. 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 $tid \cdot accept(fd)$ transition.

Socket *sid* might not be bound to a local IP address, i.e. is_1 could be $*$. In this case the socket is listening for connection attempts on port p_1 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.**

$$
\begin{array}{l}
(h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d); \\
socks := \\
socks \oplus \\
[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \\
\qquad\qquad \text{TCP\_Sock}(st, cb, \uparrow lis)))]|\!\rangle, \\
SS, MM) \\
\xrightarrow{lbl} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ECONNABORTED))_{sched\_timer}); \\
\quad socks := \\
\quad socks \oplus \\
\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \\
\qquad\qquad \text{TCP\_Sock}(st, cb, \uparrow lis)))]|\!\rangle, \\
\quad SS, MM)
\end{array}
$$

$$
\left(
\begin{pmatrix}
t = Run \wedge \\
st = LISTEN \wedge \\
cantsndmore = \mathbf{T} \wedge \\
lbl = tid\cdot accept(fd) \wedge \\
rc = fast\ fail \wedge \\
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)
\end{pmatrix}
\vee
\begin{pmatrix}
t = Accept2(sid) \wedge \\
((cantrcvmore = \mathbf{T} \wedge st = LISTEN) \vee \\
(st = CLOSED)) \wedge \\
lbl = \tau \wedge \\
rc = slow\ urgent\ fail
\end{pmatrix}
\right)
$$

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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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*·*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**

$$(h \; \langle\!| ts := ts \oplus (tid \mapsto (t)_d) |\!\rangle, SS, MM)$$
$$\xrightarrow{lbl} \; (h \; \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EINVAL))_{sched\_timer}) |\!\rangle, SS, MM)$$

$$\left(\begin{pmatrix} t = Run \wedge \\ lbl = tid \cdot accept(fd) \wedge \\ rc = fast\ fail \wedge \\ fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \text{File}(FT\_Socket(sid), ff) \end{pmatrix} \vee \begin{pmatrix} t = Accept2(sid) \wedge \\ lbl = \tau \wedge \\ rc = slow\ urgent\ fail \end{pmatrix}\right) \wedge$$
$$sid \in \mathbf{dom}(h.socks) \wedge$$
$$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$$
$$tcp\_sock.st \neq LISTEN$$

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*·*accept*(*fd*) transition and in the latter by a τ transition. In either case, the thread is left in state *Ret*(FAIL *EINVAL*) to return error *EINVAL* 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**

$$(h \; \langle\!| ts := ts \oplus (tid \mapsto (t)_d) |\!\rangle, SS, MM)$$
$$\xrightarrow{lbl} \; (h \; \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EMFILE))_{sched\_timer}) |\!\rangle, SS, MM)$$

$$\left(\begin{pmatrix} t = Run \wedge \\ lbl = tid \cdot accept(fd) \wedge \\ rc = fast\ fail \wedge \\ fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge \\ sid \in \mathbf{dom}(h.socks) \wedge \\ sock = (h.socks[sid]) \wedge \\ \text{proto\_of } sock.pr = PROTO\_TCP \end{pmatrix} \vee \begin{pmatrix} t = Accept2(sid) \wedge \\ lbl = \tau \wedge \\ rc = slow\ nonurgent\ fail \end{pmatrix}\right) \wedge$$

$\mathbf{card}(\mathbf{dom}(h.fds)) \geq OPEN\_MAX$

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·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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot accept(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge$
$sid \in \mathbf{dom}(h.socks) \wedge$
$\text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP \wedge$
$(\mathbf{if}\ bsd\_arch\ h.arch\ \mathbf{then}\ err = EINVAL$
$\mathbf{else}\ err = EOPNOTSUPP)$

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·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 *EINVAL* if *accept*() is called on a UDP socket.

7.2 *bind*() (TCP and UDP)

$bind : (fd * ip\ \mathsf{option} * port\ \mathsf{option}) \rightarrow \mathsf{unit}$

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, $bind(\_, \_, \uparrow p)$ 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, $bind(\_, \uparrow i, \_)$, the socket's local IP address is set to $\uparrow i$, 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 overrriden by setting the socket option *SO_REUSEADDR* appropriately: see bound_port_allowed (p36).

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*().

7.2.1 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: • $bind(\_, \_, \uparrow p)$ will fail with *EADDRINUSE* if another socket is bound to port p. This error may be preventable by setting the *SO_REUSEADDR* socket option. • $bind(\_, \uparrow i, \uparrow p)$ will fail with *EADDRINUSE* if another socket is bound to port p and IP address i, or is bound to port p 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. This error is never returned from a call $bind(\_, \_, *)$ that requests an autobound port.
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 *bind*() 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 *bind*() 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.

7.2.2 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*

7.2.3 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 compatilibity 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.

7.2.4 Model details

The specification only models the AF,PF_INET address families thus the address family field of the `struct sockaddr` argument to *bind*() and those errors specific to other address familes, 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.

7.2.5 Summary

bind_1	**all: fast succeed**	Successfully assign a local address to a socket (possibly by autobinding the port)
bind_2	**all: fast fail**	Fail with *EADDRINUSE*: the specified address is already in use
bind_3	**all: fast fail**	Fail with *EADDRNOTAVAIL*: the specified IP address is not available on the host
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
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
bind_9	**all: fast badfail**	Fail with *ENOBUFS*: no ephemeral ports free for autobinding or, on WinXP only, insufficient buffers available.

7.2.6 Rules

bind_1 **all: fast succeed** **Successfully assign a local address to a socket (possibly by autobinding the port)**

$$(h_0, SS, MM) \xrightarrow{tid \cdot bind(fd, is_1, ps_1)} (h, SS, MM)$$

$$
\begin{aligned}
& h_0 = h' \,⦇\, ts := ts \oplus (tid \mapsto (Run)_d); \\
& \quad socks := socks \oplus \\
& \quad [(sid, \textsc{Sock}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, pr))] \\
& \quad ⦈ \wedge \\
& h = h' \,⦇\, ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}); \\
& \quad socks := socks \oplus \\
& \quad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, pr))]; \\
& \quad bound := bound ⦈ \wedge \\
& fd \in \mathbf{dom}(h_0.fds) \wedge \\
& fid = h_0.fds[fd] \wedge \\
& h_0.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge \\
& sid \notin (\mathbf{dom}(socks)) \wedge \\
& (\forall i_1. is_1 = \uparrow i_1 \implies i_1 \in local\_ips(h_0.ifds)) \wedge \\
& p_1 \in \text{autobind}(ps_1, (\text{proto\_of } pr), h_0, socks) \wedge \\
& bound = sid :: h_0.bound \wedge \\
& (h_0.privs \vee p_1 \notin \text{privileged\_ports } h_0) \wedge \\
& \text{bound\_port\_allowed } pr (h_0.socks \backslash\backslash sid) sf\ h_0.arch\ is_1\ p_1 \wedge \\
& (\mathbf{case}\ pr\ \mathbf{of} \\
& \quad \text{TCP\_PROTO}(tcp\_sock) \rightarrow tcp\_sock = \text{TCP\_Sock0}(CLOSED, cb, *) \wedge \\
& \qquad (bsd\_arch\ h_0.arch \implies cantsndmore = \mathbf{F}) \parallel \\
& \quad \text{UDP\_PROTO}(udp\_sock) \rightarrow udp\_sock = \text{UDP\_Sock0}([\,]))
\end{aligned}
$$

Description

The call $bind(fd, is_1, ps_1)$ is perfomed 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, $i_1 \in local\_ips(h_0.ifds)$.

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 $h_0.priv = \mathbf{T}$.

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 (p36) for further information.

The call proceeds by performing a $tid \cdot bind(fd, is_1, ps_1)$ 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot bind(fd, is_1, \uparrow p_1)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EADDRINUSE))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = (h.socks[sid]) \wedge$
$\neg(\text{bound\_port\_allowed}\ sock.pr(h.socks \backslash\backslash sid) sock.sf\ h.arch\ is_1\ p_1) \wedge$
$(\mathbf{option\_case}\ \mathbf{T}\ (\lambda i_1.i_1 \in local\_ips(h.ifds))\ is_1 \vee windows\_arch\ h.arch)$

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 bound_port_allowed (p36) for details.

The call proceeds by a $tid \cdot bind(fd, is_1, \uparrow p_1)$ transition leaving the thread in state $Ret(\text{FAIL } EADDRINUSE)$ to return error *EADDRINUSE* to the caller.

bind_3 **<u>all: fast fail</u> Fail with *EADDRNOTAVAIL*: the specified IP address is not available on the host**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot bind(fd, \uparrow i_1, ps_1)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EADDRNOTAVAIL))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$i_1 \notin local\_ips(h.ifds)$

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, $i_1 \notin local\_ips(h.ifds)$, and therefore can not be assigned to the socket.

The call proceeds by a $tid \cdot bind(fd, \uparrow i_1, ps_1)$ transition leaving the thread in state $Ret(\text{FAIL } EADDRNOTAVAIL)$ to return error *EADDRNOTAVAIL* to the caller.

bind_5 **<u>all: fast fail</u> 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot bind(fd, is_1, ps_1)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EINVAL))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$h.socks[sid] = sock \wedge$
$(sock.ps_1 \neq * \vee$
$(bsd\_arch\ h.arch \wedge sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\quad (sock.cantsndmore \vee$
$\quad \mathbf{T})))$

Description From thread *tid*, which is in the *Run* state, a $bind(fd, is_1, ps_1)$ 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 $tid \cdot bind(fd, is_1, ps_1)$ transition is made, leaving the thread state $Ret(\text{FAIL } EINVAL)$.

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 **<u>all: fast fail</u> Fail with *EACCES*: the specified port is priveleged and the current process does not have permission to bind to it**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$

$\xrightarrow{tid \cdot bind(fd, is_1, \uparrow p_1)} \quad (h \; \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EACCES))_{sched\_timer}) \rangle\!\rangle, SS, MM)$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$(\neg h.privs \wedge p_1 \in \text{privileged\_ports } h)$

Description

From thread *tid*, which is in the *Run* state, a $bind(fd, is_1, \uparrow p_1)$ 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, $p_1 \in$ 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 $tid \cdot bind(fd, is_1, \uparrow p_1)$ transition leaving the thread in state $Ret(\text{FAIL } EACCES)$ 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.**

$(h \; \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d) \rangle\!\rangle, SS, MM)$

$\xrightarrow{tid \cdot bind(fd, is_1, ps_1)} \quad (h \; \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } ENOBUFS))_{sched\_timer}) \rangle\!\rangle, SS, MM)$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$ps_1 = * \wedge$
$((\text{autobind}(ps_1, (\text{proto\_of}(h.socks[sid]).pr), h, h.socks) = \emptyset) \vee$
$windows\_arch \; h.arch)$

Description

From thread *tid*, which is in the *Run* state, a $bind(fd, is_1, ps_1)$ call is made where *fd* refers to a socket *sid*.

A port is not specifed 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 $tid \cdot bind(fd, is_1, ps_1)$ transition leaving the thread state $Ret(\text{FAIL } ENOBUFS)$ 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.

7.3 *close*() (TCP and UDP)

$close : fd \rightarrow \mathsf{unit}$

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 asychronously, 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.

7.3.1 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 *close*() 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.

7.3.2 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*

7.3.3 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.

7.3.4 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.

7.3.5 Summary

close_1	**all: fast succeed**	Successfully close a file descriptor that is not the last file descriptor for a socket
close_2	**tcp: fast succeed**	Successfully perform a graceful close on the last file descriptor of a synchronised socket
close_3	**tcp: fast succeed**	Successful abortive close of a synchronised socket
close_4	**tcp: block**	Block on a lingering close on the last file descriptor of a synchronised socket
close_5	**tcp: slow urgent succeed**	Successful completion of a lingering close on a synchronised socket
close_6	**tcp: slow nonurgent fail**	Fail with *EAGAIN*: unsuccessful completion of a lingering close on a synchronised socket
close_7	**tcp: fast succeed**	Successfully close the last file descriptor for a socket in the *CLOSED*, *SYN_SENT* or *SYN_RECEIVED* states.
close_8	**tcp: fast succeed**	Successfully close the last file descriptor for a listening TCP socket
close_10	**udp: fast succeed**	Successfully close the last file descriptor of a UDP socket

7.3.6 Rules

close_1 **all: fast succeed** **Successfully close a file descriptor that is not the last file descriptor**

for a socket

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ fds := fds|\!\rangle,\ SS, MM) \xrightarrow{tid\cdot close(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ fds := fds'|\!\rangle,\ SS, MM)$$

$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$fid\_ref\_count(fds, fid) > 1 \wedge$
$fds' = fds \backslash\backslash fd$

Description

A $close(fd)$ call is performed where fd refers to either a TCP or UDP socket. At least two file descriptors refer to file description fid, $fid\_ref\_count(fds, fid) > 1$, of which one is fd, $fid = fds[fd]$.

The $close(fd)$ call proceeds by a $tid\cdot close(fd)$ 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 $fds' = fds\backslash\backslash fd$, effectively reducing the reference count of the file description by one. The $close()$ call does not alter the socket's state as other file descriptors still refer to the socket through file description fid.

close_2 <u>**tcp: fast succeed**</u> **Successfully perform a graceful close on the last file descriptor of a synchronised socket**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ fds := fds;\\ files := files \oplus\\ \quad [(fid, \textsc{File}(FT\_Socket(sid), ff))];\\ socks := socks \oplus\\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,\\ \qquad \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)\end{array}$$

$$\xrightarrow{tid\cdot close(fd)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\\ fds := fds';\\ files := files\backslash\backslash fid;\\ socks := socks \oplus\\ \quad [(sid, \textsc{Sock}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},\\ \qquad \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)\end{array}$$

$(st \in \{ESTABLISHED; FIN\_WAIT\_1; CLOSING; FIN\_WAIT\_2;$
$\quad TIME\_WAIT; CLOSE\_WAIT; LAST\_ACK\} \vee$
$st = SYN\_RECEIVED \wedge linux\_arch\ h.arch) \wedge$
$(sf.t(SO\_LINGER) = \infty \vee$
$ff.b(O\_NONBLOCK) = \mathbf{T} \wedge sf.t(SO\_LINGER) \neq 0 \wedge \neg linux\_arch\ h.arch) \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$fid\_ref\_count(fds, fid) = 1 \wedge$
$fds' = fds\backslash\backslash fd \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$(peek, inline) = (\mathbf{F}, \mathbf{T}) \wedge$
$\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, data) s\ s'$

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, $sf.t(SO\_LINGER) = \infty$, and regardless of whether the socket is in non-blocking mode or not, i.e. $ff.b(O\_NONBLOCK)$ is unconstrained, the call to $close()$ proceeds successfully without blocking.

On all platforms except for Linux, if the socket is in non-blocking mode $ff.b(O\_NONBLOCK) = \mathbf{T}$ the linger option may be set with a positive duration: $sf.t(SO\_LINGER) \neq 0$). 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 $tid \cdot close(fd)$ 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 $fds' = fds \backslash\backslash fd$ and the file description entry fid is removed from the finite map of file descriptors $files \backslash\backslash fid$. The socket entry itself, $(sid, \textsc{Sock}(\uparrow fid, \ldots,))$ 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, $ff.b(O\_NONBLOCK) = \mathbf{T}$ and a linger option is set to a non-zero value, $sf.t(SO\_LINGER) \neq 0$, the socket may block on a call to $close()$. See also $close\_4$ (p75).

close_3 **tcp: fast succeed** **Successful abortive close of a synchronised socket**

$$(h \ \langle\!| ts := ts \oplus (tid \mapsto (Run)_d); fds := fds; files := files \oplus [(fid, \textsc{File}(FT\_Socket(sid), ff))]; socks := socks \oplus [(sid, sock)]; oq := oq |\!\rangle, SS, MM) \xrightarrow{tid \cdot close(fd)} (h \ \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}); fds := fds'; files := files; socks := socks \oplus [(sid, sock')]; oq := oq' |\!\rangle, SS'', MM)$$

$(st \in \{ESTABLISHED; FIN\_WAIT\_1; CLOSING; FIN\_WAIT\_2;$
$\quad TIME\_WAIT; CLOSE\_WAIT; LAST\_ACK\} \vee$
$st = SYN\_RECEIVED \wedge linux\_arch\ h.arch) \wedge$
$sock = \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *)) \wedge$
$sf.t(SO\_LINGER) = 0 \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$fid\_ref\_count(fds, fid) = 1 \wedge$
$fds' = fds \backslash\backslash fd \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin \mathbf{dom}(socks) \wedge$
$sock' = (\text{tcp\_close}\ h.arch\ sock)\langle\!| fid := * |\!\rangle \wedge$
$oflgs = oflgs \ \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{T} |\!\rangle \wedge$

Rule version: \$ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp \$

$odata \in UNIV \wedge$
$SS = SS_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata) s\ s' \wedge$
$SS' = SS_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')] \wedge$
$\text{destroy}(i_1, p_1, i_2, p_2) SS'\ SS''$

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 >= *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, $sf.t(SO\_LINGER) = 0$, to signify that an abortive closure of socket *sid* is required.

The *close(fd)* call proceeds by a *tid·close(fd)* transition leaving the host in the successful return state *Ret(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 enqueue_and_ignore_fail (p50), to create new output queue *oq'*. The enqueue_and_ignore_fail function always succeeds; if it is not possible to add the reset segment *seq* 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 $fds' = fds \backslash\backslash fd$ 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 (p52). 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 ≥ *ESTABLISHED*.

close_4 **tcp: block** **Block on a lingering close on the last file descriptor of a synchronised socket**

$$
\begin{array}{l}
(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\
\quad fds := fds;\\
\quad files := files \oplus\\
\qquad [(fid, \text{FILE}(FT\_Socket(sid), ff))];\\
\quad socks := socks \oplus\\
\qquad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\\
\quad SS, MM)\\
\xrightarrow{tid\cdot close(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Close2(sid))_{slow\_timer(sf.t(SO\_LINGER))});\\
\qquad fds := fds';\\
\qquad files := files;\\
\qquad socks := socks \oplus\\
\qquad\quad [(sid, \text{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\\
\qquad SS, MM)
\end{array}
$$

$(st \in \{ESTABLISHED; FIN\_WAIT\_1; CLOSING; FIN\_WAIT\_2;$
$\quad TIME\_WAIT; CLOSE\_WAIT; LAST\_ACK\} \vee$
$st = SYN\_RECEIVED \wedge linux\_arch\ h.arch) \wedge$
$sf.t(SO\_LINGER) \notin \{0; \infty\} \wedge$

$(\mathit{ff}.b(\mathit{O\_NONBLOCK}) = \mathbf{F} \vee (\mathit{ff}.b(\mathit{O\_NONBLOCK}) = \mathbf{T} \wedge \mathit{linux\_arch}\ h.\mathit{arch})) \wedge$
$\mathit{fd} \in \mathbf{dom}(\mathit{fds}) \wedge$
$\mathit{fid} = \mathit{fds}[\mathit{fd}] \wedge$
$\mathit{fid\_ref\_count}(\mathit{fds}, \mathit{fid}) = 1 \wedge$
$\mathit{fds}' = \mathit{fds} \backslash\backslash \mathit{fd} \wedge$
$\mathit{fid} \notin (\mathbf{dom}(\mathit{files}))$

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: $\mathit{fid\_ref\_count}(\mathit{fds}, \mathit{fid}) = 1$. The TCP socket *sid* has a blocking mode of operation, $\mathit{ff}.b(\mathit{O\_NONBLOCK}) = \mathbf{F}$, and is in a synchronised state, i.e. a state $\geq \mathit{ESTABLISHED}$.

On Linux, the socket is also permitted to be in the *SYN_RECEIVED* state and it may have non-blocking semantics $\mathit{ff}.b(\mathit{O\_NONBLOCK}) = \mathbf{T}$, 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), $\mathit{sf}.t(\mathit{SO\_LINGER}) \notin \{0; \infty\}$. 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 $close(fd)$ call proceeds by a $tid \cdot close(fd)$ transition leaving the host in the blocked state $\mathit{Close2}(sid)$. The socket is marked as unable to send and receive further data, $\mathit{cantsndmore} = \mathbf{T} \wedge \mathit{cantrcvmore} = \mathbf{T}$; 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 $\mathit{fds}' = \mathit{fds} \backslash\backslash \mathit{fd}$ and file description entry *fid* is removed from the finite map of file descriptors. The socket entry itself, $(sid, \mathrm{SOCK}(\uparrow \mathit{fid}, \ldots))$, is not destroyed at this point; it remains until the TCP socket has been successfully closed by future asychronous events.

Variations

Linux	The socket can be in the *SYN_RECEIVED* state or in one of the synchronised states $\geq \mathit{ESTABLISHED}$. On Linux, non-blocking semantics do not take precedence over the *SO_LINGER* option, i.e. if the socket is non-blocking, $\mathit{ff}.b(\mathit{O\_NONBLOCK}) = \mathbf{T}$ and a linger option is set to a non-zero value, $\mathit{sf}.t(\mathit{SO\_LINGER}) \neq 0$ the socket may block on a call to $close()$.

close_5 **tcp: slow urgent succeed** **Successful completion of a lingering close on a synchronised socket**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathit{Close2}(sid))_d); \mathit{socks} := \mathit{socks} \oplus [(sid, \mathrm{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \mathrm{TCP\_Sock}(st, cb, *)))]|\!\rangle, SS, MM)$$

$$\xrightarrow{\tau}$$

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathit{Ret}(\mathit{OK}()))_{\mathit{sched\_timer}}); \mathit{socks} := \mathit{socks} \oplus [(sid, \mathrm{SOCK}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \mathrm{TCP\_Sock}(st, cb, *)))]|\!\rangle, SS, MM)$$

$st \in \{\mathit{TIME\_WAIT}; \mathit{CLOSED}; \mathit{FIN\_WAIT\_2}\}$

Description

A previous call to $close()$ with the linger option set on the socket blocked leaving thread *tid* in the $\mathit{Close2}(sid)$ state. The socket *sid* has successfully transmitted all the data in its send queue, $sndq = [\,]$, and has completed a graceful close of the connection: $st \in \{\mathit{TIME\_WAIT}; \mathit{CLOSED}; \mathit{FIN\_WAIT\_2}\}$.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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 asychronous 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**

$$(h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Close2(sid))_d);\ socks := socks \oplus [(sid, sock)]\ \rbrack\!\rangle,\ SS, MM) \xrightarrow{\tau} (h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EAGAIN))_{sched\_timer});\ socks := socks \oplus [(sid, sock)]\ \rbrack\!\rangle,\ SS, MM)$$

$sock = \mathrm{Sock}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T}, \mathrm{TCP\_Sock}(st, cb, *)) \wedge$
$timer\_expires\ d \wedge$
$st \notin \{TIME\_WAIT; CLOSED\}$

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: $st \notin \{TIME\_WAIT; CLOSED\}$.

The rule proceeds via a τ transition leaving thread *tid* in the $Ret(\mathrm{FAIL}\ EAGAIN)$ 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 asychronous events.

The asychronous 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. $st \notin \{TIME\_WAIT; CLOSED\}$. 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.**

$$(h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Run)_d);\ fds := fds;\ files := files \oplus [(fid, \mathrm{File}(FT\_Socket(sid), ff))];\ socks := socks \oplus [(sid, sock)]\rbrack\!\rangle,\ SS, MM) \xrightarrow{tid \cdot close(fd)} (h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ fds := fds';\ files := files;\ socks := socks\rbrack\!\rangle,\ SS', MM)$$

$(tcp\_sock.st \in \{CLOSED; SYN\_SENT\} \vee$
$tcp\_sock.st = SYN\_RECEIVED \wedge \neg linux\_arch\ h.arch) \wedge$
$\mathrm{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin (\mathbf{dom}(socks)) \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$

$fid\_ref\_count(fds, fid) = 1 \wedge$
$fds' = fds \backslash\backslash fd \wedge$

case $tcp\_sock.st \in \{CLOSED; LISTEN\}$ **of**
T $\rightarrow SS' = SS$
$\|$ **F** $\rightarrow$ **if** exists_quad_of *sock* **then**
destroy(quad_of *sock*)$SS\ SS'$
else $SS' = SS$

Description

A $close(fd)$ 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: $fid\_ref\_count(fds, fid) = 1$. The TCP socket *sock* is not in a synchronised state: $st \in \{CLOSED; SYN\_SENT\}$.

The $close(fd)$ call proceeds by a $tid \cdot close(fd)$ transition leaving the host in the successful return state $Ret(OK())$.

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 *fid* 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 $close()$ 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**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$fds := fds;$
$files := files \oplus [(fid, \text{FILE}(FT\_Socket(sid), ff))];$
$socks := socks \oplus [(sid, sock)];$
$listen := listen;$
$oq := oq|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot close(fd)}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});$
$fds := fds';$
$files := files;$
$socks := socks';$
$listen := listen';$
$oq := oq'|\!\rangle,$
$SS'', MM)$

$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,$
$\text{TCP\_Sock}(LISTEN, cb, \uparrow lis)) \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$fid\_ref\_count(fds, fid) = 1 \wedge$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin (\mathbf{dom}(socks)) \wedge$

(* 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 *)
$(bsd\_arch\ h.arch \vee (cantsndmore = \mathbf{F} \wedge cantrcvmore = \mathbf{F})) \wedge$

(* BSD and Linux do not send RSTs to sockets on $lis.q0$. *)
$socks\_to\_rst = \{sock' \mid \exists sid'\ tcp\_sock'.sid' \in lis.q \wedge$
$sock' = socks[sid'] \wedge$
$\text{TCP\_PROTO}(tcp\_sock') = sock'.pr \wedge$
$tcp\_sock'.st \notin \{CLOSED; LISTEN; SYN\_SENT\}\} \wedge$

$\mathbf{dom}(SS') = \mathbf{dom}(SS) \wedge$

$(\forall sock'.sock' \in socks\_to\_rst \implies$
$\quad \mathbf{let}\ (i_1, p_1, i_2, p_2) = \text{quad\_of}\ sock'\ \mathbf{in}$
$\quad \mathbf{let}\ streamid = \text{streamid\_of\_quad}(i_1, p_1, i_2, p_2)\ \mathbf{in}$
$\quad \exists oflgs\ odata.$
$\quad oflgs = oflgs\ (\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{T} |\!) \wedge$
$\quad odata \in UNIV \wedge$
$\quad \text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)(SS[streamid])(SS'[streamid])) \wedge$

$(\forall streamid :: \mathbf{dom}(SS).$
$\quad \neg(streamid \in (\mathbf{image}(\text{streamid\_of\_quad}\ o\ \text{quad\_of})socks\_to\_rst)) \implies$
$\quad SS'[streamid] = SS[streamid]) \wedge$

$fds' = fds \backslash\backslash fd \wedge$
$listen' = \mathbf{filter}(\lambda sid'.sid' \neq sid)listen \wedge$
$socks' = socks|_{\{sid' | sid' \notin lis.q0 @ lis.q\}} \wedge$

(* $removed\_sids$ does not include sid *)
$\mathbf{let}\ removed\_sids = \{sid' \mid sid' \in lis.q0\ @\ lis.q\}\ \mathbf{in}$
$\mathbf{let}\ removed\_socks = \{sock\} \cup \{sock' \mid \exists sid'.sid' \in removed\_sids \wedge$
$socks[sid'] = sock'\}\ \mathbf{in}$
$\mathbf{let}\ destroyed = \{(i_1, p_1, i_2, p_2) \mid \exists sock.sock \in removed\_socks \wedge$
$(sock.is_1, sock.ps_1, sock.is_2, sock.ps_2) = (\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)\}\ \mathbf{in}$

(* Some streams are destroyed *)
$\text{destroy\_quads}\ destroyed\ SS'\ SS''$

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*, $fid\_ref\_count(fds, fid) = 1$. 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.q0*. 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 seqments *rsts_to_go* is created for each of the sockets referenced by both queues. This is performed by looking up each socket $sock'$ for every sid' in the concatentation of both queues, *lis.q0* @ *lis.q*, and extracting their address quads $(sock'.is_1, sock'.is_2, sock'.ps_1, sock'.ps_2)$ and control blocks *cb*.

The $close(fd)$ call proceeds by a $tid{\cdot}close(fd)$ transition leaving the host in the successful return state $Ret(OK())$.

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**

$$
\begin{array}{l}
(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\
fds := fds;\\
files := files \oplus [(fid, \text{FILE}(FT\_Socket(sid), ff))];\\
socks := socks \oplus\\
\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \text{UDP\_PROTO}(udp)))]|\!\rangle,\\
SS, MM)\\
\xrightarrow{tid\cdot close(fd)} \quad (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\\
\qquad fds := fds';\\
\qquad files := files;\\
\qquad socks := socks|\!\rangle,\\
\qquad SS, MM)
\end{array}
$$

$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$fid\_ref\_count(fds, fid) = 1 \wedge$
$fds' = fds \backslash\backslash fd \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin (\mathbf{dom}(socks))$

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 *close*(*fd*) call is made and succeeds.

A $tid\cdot close(fd)$ transition is made, leaving the thread state $Ret(OK())$. The socket *sid* is removed from the host's finite map of sockets $socks \oplus \ldots$, the file description record indexed by *fid* is removed from the host's finite map of file descriptions $files \oplus \ldots$, and *fd* is removed from the host's finite map of file descriptors $fds' = fds \backslash\backslash fd$.

7.4 *connect*() (TCP and UDP)

$connect : fd * ip * port$ option $\rightarrow$ unit

A call to $connect(fd, ip, port)$ 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 *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 sequeunce 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 *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 *connect*() call will fail with an *EINPROGRESS* error (or *EALREADY* on WinXP), and connection establishment will proceed asynchronously.

Calling *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, $pselect([\,],[fd],[\,],*,\_)$ 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*.

7.4.1 Errors

A call to *connect*() can fail with the errors below, in which case the corresponding exception is raised:

EADDRNOTAVAIL	There is no route to the peer; a port must be specified ($port \neq *$); or there are no ephemeral ports left.
EADDRINUSE	The address quad that would result if the connection was successful is in use by another socket of the same protocol.
EAGAIN	On WinXP, the socket is non-blocking and the connection cannot be established immediately: it will be established asynchronously. [TCP ONLY]
EALREADY	A connection attempt is already in progress on the socket but not yet complete: it is in state *SYN_SENT* or *SYN_RECEIVED*. [TCP ONLY]
ECONNREFUSED	Connection rejected by peer. [TCP ONLY]

ECONNRESET	Connection rejected by peer. [TCP ONLY]
EHOSTUNREACH	No route to the peer.
EINPROGRESS	The socket is non-blocking and the connection cannot be established immediately: it will be established asynchronously. [TCP ONLY]
EINVAL	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.

7.4.2 Common cases

TCP: *socket_1*; *connect_1*; ...
UDP: *socket_1*; *bind_1*; *connect_8*; ...

7.4.3 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, constr 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 $port = \uparrow p$; 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.

7.4.4 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 seperately.
- `EACCES`, `ELOOP`, and `ENAMETOOLONG` are errors dealing with Unix domain sockets which are not modelled here.

7.4.5 Summary

connect_1	**tcp: rc**	Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue
connect_1a	**tcp: rc**	Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue
connect_2	**tcp: slow urgent succeed**	Successfully return from blocking state after connection is successfully established
connect_3	**tcp: slow urgent fail**	Fail with the pending error on a socket in the *CLOSED* state
connect_4	**tcp: slow urgent fail**	Fail: socket has pending error
connect_4a	**tcp: fast fail**	Fail with pending error
connect_5	**tcp: fast fail**	Fail with *EALREADY*, *EINVAL*, *EISCONN*, *EOPNOTSUPP*: socket already in use
connect_5a	**all: fast fail**	Fail: no route to host
connect_5b	**all: fast fail**	Fail with *EADDRINUSE*: address already in use
connect_5c	**all: fast fail**	Fail with *EADDRNOTAVAIL*: no ephemeral ports left
connect_5d	**tcp: block**	Block, entering state *Connect2*: connection attempt already in progress and connect called with blocking semantics
connect_6	**tcp: fast fail**	Fail with *EINVAL*: socket has been shutdown for writing
connect_7	**udp: fast succeed**	Set peer address on socket with binding quad $*, ps_1, *, *$
connect_8	**udp: fast succeed**	Set peer address on socket with local address set
connect_9	**udp: fast fail**	Fail with *EADDRNOTAVAIL*: port must be specified in *connect*() call on FreeBSD
connect_10	**udp: fast fail**	Fail with pending error on FreeBSD, but still set peer address

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

7.4.6 Rules

connect_1 **tcp: rc** **Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue**

$$(h, SS, MM) \xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} (h', SS', MM)$$

(* Thread *tid* is in state *Run* and TCP socket *sid* has binding quad (is_1, ps_1, is_2, ps_2). *)

$$\begin{aligned}
h = h_0 \,\langle\!|\, & ts := ts_ \oplus (tid \mapsto (Run)_d);\\
& socks := socks \oplus \\
& \quad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
& \qquad \text{TCP_Sock}(st, cb, *)))];\\
& oq := oq \,|\!\rangle \wedge
\end{aligned}$$

(* Thread *tid* ends in state t' with updated host sockets and output queue *)

$$\begin{aligned}
h' = h_0 \,\langle\!|\, & ts := ts_ \oplus (tid \mapsto t');\\
& socks := socks \oplus \\
& \quad [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1', \uparrow p_1', is_2', ps_2', es'', \mathbf{F}, \mathbf{F},\\
& \qquad \text{TCP_Sock}(st', cb''', *)))];\\
& bound := bound;\\
& oq := oq' \,|\!\rangle \wedge
\end{aligned}$$

(* File descriptor *fd* refers to TCP socket *sid* *)

$$\begin{aligned}
& fd \in \mathbf{dom}(h_0.fds) \wedge \\
& fid = h_0.fds[fd] \wedge \\
& h_0.files[fid] = \textsc{File}(FT_Socket(sid), ff) \wedge
\end{aligned}$$

(* 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. *)

$$i_1' \in auto\_outroute(i_2, is_1, h.rttab, h.ifds) \wedge$$

(* 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' *)

$$p_1' \in \text{autobind}(ps_1, PROTO\_TCP, h, h.socks) \wedge$$

(* If autobinding occurs then *sid* is added to the head of the host's list of bound sockets. *)

$$(\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h.bound\ \mathbf{else}\ bound = h.bound) \wedge$$

(* 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*. *)

$$\begin{aligned}
(& st = CLOSED \wedge is_2 = * \wedge ps_2 = * \wedge \\
& es = * \wedge cantsndmore = \mathbf{F} \wedge (cantrcvmore = \mathbf{F} \vee bsd_arch\ h.arch)) \wedge
\end{aligned}$$

(* No other TCP sockets on the host have the address quad $(\uparrow i_1', \uparrow p_1', \uparrow i_2, \uparrow p_2)$. *)

$$\begin{aligned}
\neg(\exists (sid', s) :: (h.socks \backslash\backslash sid).\ & s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge \\
& s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge \\
& \text{proto_of}\ s.pr = PROTO_TCP) \wedge
\end{aligned}$$

$cb' = cb \wedge$

(* 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_flgs_data (p262) for details. *)
$(oflgs, odata) \in$ make_syn_flgs_data $\wedge$

(* and send it out... *)

(* If possible, enqueue the segment *seg* on the host's outqueue. The auxiliary function *stream_rollback_tcp_output* (p49) 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. *)
$\exists outsegs'.$
$stream\_rollback\_tcp\_output\ \mathbf{F}(\uparrow i_1', \uparrow i_2) h.arch\ h.rttab\ h.ifds\ cb'(cb'', es', outsegs') \wedge$
$cb''' = (\mathbf{if}\ (outsegs' \vee windows\_arch\ h.arch)\ \mathbf{then}\ \ cb''\ \ \mathbf{else}\ \ cb) \wedge$
$(INFINITE\_RESOURCES \implies queued) \wedge$

(* 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 *)
$((\neg ff.b(O\_NONBLOCK) \vee (bsd\_arch\ h.arch \wedge i_2\ \in\ local\_ips\ h.ifds)) \wedge$
$t' = (Connect2(sid))_{never\_timer} \wedge rc = block \wedge$
$es'' = es' \wedge st' = SYN\_SENT \wedge is_2' = \uparrow i_2 \wedge ps_2' = \uparrow p_2 \wedge$
$s = \mathrm{initial\_streams}(i_1', p_1', i_2, p_2) \wedge$
$\mathrm{write}(i_1', p_1', i_2, p_2)(oflgs, odata) s\ s' \wedge$
$SS' = SS \oplus [(\mathrm{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s')]) \vee$
(* non-blocking socket *)
$(ff.b(O\_NONBLOCK) \wedge$
$es = * \wedge$
$(err = (\mathbf{if}\ \ windows\_arch\ h.arch\ \ \mathbf{then}\ \ EAGAIN\ \ \mathbf{else}\ \ EINPROGRESS) \vee \uparrow err = es') \wedge$
$t' = (Ret(\mathrm{FAIL}\ err))_{sched\_timer} \wedge rc = fast\ fail \wedge es'' = * \wedge$
$\mathbf{if}\ \neg queued\ \ \mathbf{then}$
$st' = CLOSED \wedge is_2' = * \wedge ps_2' = * \wedge$
(* under BSD *st* could be *TIME_WAIT* *)
(* REMARK this fail quick behaviour breaks abstraction boundaries *)
$SS' = SS$
else

$st' = SYN\_SENT \wedge is_2' = \uparrow i_2 \wedge ps_2' = \uparrow p_2 \wedge$
$s = \text{initial\_streams}(i_1', p_1', i_2, p_2) \wedge$
$\text{write}(i_1', p_1', i_2, p_2)(oflgs, odata) s\ s' \wedge$
$SS' = SS \oplus [(\text{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s')])$
)

Description

From thread *tid*, a $connect(fd, i_2, \uparrow p_2)$ 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $Connect2(sid)$. If the socket is non-blocking (its *O_NONBLOCK* flag is set) and the segment enqueuing was successful then the call will fail: a $tid \cdot connect(fd, i_2, \uparrow p_2)$ 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ 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 *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 *connect*() call is made.
WinXP	If there is an early buffer allocation failure when enqueuing the segment, then it will not be placed on the host's out-queue and $es' = ENOBUFS$; the socket's control block will be cb' with its *snd_nxt* and *snd_max* fields set to the intial sequence number, its *last_ack_seen* and *rcv_adv* fields set to 0, its *tt_delack* option set to $*$, its *tt_rexmt* 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 case a loopback packet to be sent on the wire then the socket's control block will be cb'.

connect_1a **tcp: rc Begin connection establishment by creating a SYN and trying to enqueue it on host's outqueue**

$$(h, SS, MM) \xrightarrow{lbl} (h', SS', MM)$$

(* Thread *tid* is in state *Run* and TCP socket *sid* has binding quad (is_1, ps_1, is_2, ps_2). *)
$h = h_0 \llparenthesis ts := ts\_ \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \text{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
$\text{TCP\_Sock}(st, cb, *)))];$
$oq := oq \rrparenthesis \wedge$

(* Thread *tid* ends in state t' with updated host sockets and output queue *)

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$h' = h_0$ ⦇ $ts := ts\_ \oplus (tid \mapsto t');$
$socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1', \uparrow p_1', is_2', ps_2', es'', \mathbf{F}, \mathbf{F},$
$\text{TCP\_Sock}(st', cb''', *)))];$
$bound := bound;$
$oq := oq'$ ⦈ $\wedge$

(* File descriptor fd refers to TCP socket sid *)
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$

(* 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. *)

$i_1' \in auto\_outroute(i_2, is_1, h.rttab, h.ifds) \wedge$
(* 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' *)
$p_1' \in \text{autobind}(ps_1, PROTO\_TCP, h, h.socks) \wedge$
(* If autobinding occurs then sid is added to the head of the host's list of bound sockets. *)
$(\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h.bound\ \mathbf{else}\ bound = h.bound) \wedge$

(* 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. *)

$(bsd\_arch\ h.arch \wedge st = TIME\_WAIT \wedge$
$(is_2 \neq * \implies ps_2 \neq *) \wedge$
$(ps_2 \neq * \implies is_2 \neq *)) \wedge$

(* No other TCP sockets on the host have the address quad $(\uparrow i_1', \uparrow p_1', \uparrow i_2, \uparrow p_2)$. *)
$\neg(\exists(sid', s) :: (h.socks \backslash\backslash sid).$
$s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge$
$s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge$
$\text{proto\_of}\ s.pr = PROTO\_TCP) \wedge$

$cb' = cb \wedge$

(* 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_flgs_data (p262) for details. *)
$(oflgs, odata) \in \text{make\_syn\_flgs\_data} \wedge$

(* and send it out... *)

(* If possible, enqueue the segment seg on the host's outqueue. The auxiliary function rollback_tcp_output (p48) 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. *)
$\exists outsegs'.$
$stream\_rollback\_tcp\_output\ \mathbf{F}(\uparrow i_1', \uparrow i_2) h.arch\ h.rttab\ h.ifds\ cb'(cb'', es', outsegs') \wedge$
$cb''' = (\mathbf{if}\ (outsegs' \vee windows\_arch\ h.arch)\ \mathbf{then}\ cb''\ \mathbf{else}\ cb) \wedge$

$(\mathit{INFINITE\_RESOURCES} \implies \mathit{queued}) \wedge$

(* 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 *)
$((\neg \mathit{ff}.b(\mathit{O\_NONBLOCK}) \vee (\mathit{bsd\_arch}\ h.\mathit{arch} \wedge i_2 \in \mathit{local\_ips}\ h.\mathit{ifds})) \wedge$
$\quad t' = (\mathit{Connect2}(\mathit{sid}))_{\mathit{never\_timer}} \wedge \mathit{rc} = \mathit{block} \wedge$
$\quad \mathit{es}'' = \mathit{es}' \wedge \mathit{st}' = \mathit{SYN\_SENT} \wedge \mathit{is}_2' = \uparrow i_2 \wedge \mathit{ps}_2' = \uparrow p_2 \wedge$
$\quad\quad$ (* BSD and $st = \mathit{TIME\_WAIT}$, so new new stream created *)
$\quad\quad \mathit{lbl} = \mathit{tid}\cdot\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2) \wedge$
$\quad\quad \mathit{SS} = \mathit{SS}_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s)] \wedge$
$\quad\quad \mathrm{write}(i_1', p_1', i_2, p_2)(\mathit{oflgs}, \mathit{odata})s\ s' \wedge$
$\quad\quad \mathit{SS}' = \mathit{SS}_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s)]) \vee$
(* non-blocking socket *)
$(\mathit{ff}.b(\mathit{O\_NONBLOCK}) \wedge$
$\quad \mathit{es} = * \wedge$
$\quad (\mathit{err} = (\mathbf{if}\ \mathit{windows\_arch}\ h.\mathit{arch}\ \mathbf{then}\ \mathit{EAGAIN}\ \mathbf{else}\ \mathit{EINPROGRESS}) \vee \uparrow \mathit{err} = \mathit{es}') \wedge$
$\quad t' = (\mathit{Ret}(\mathrm{FAIL}\ \mathit{err}))_{\mathit{sched\_timer}} \wedge \mathit{rc} = \mathit{fast\ fail} \wedge \mathit{es}'' = * \wedge$
$\quad \mathbf{if}\ \neg \mathit{queued}\ \mathbf{then}$
$\quad\quad \mathit{st}' = \mathit{CLOSED} \wedge \mathit{is}_2' = * \wedge \mathit{ps}_2' = * \wedge$
$\quad\quad$ (* under BSD $st = \mathit{TIME\_WAIT}$, and we destroy a stream *)
$\quad\quad$ (* REMARK this fail quick behaviour breaks abstraction boundaries *)

$\quad\quad \exists i_1\ p_1.(\uparrow i_1, \uparrow p_1) = (\mathit{is}_1, \mathit{ps}_1) \wedge$
$\quad\quad \mathrm{destroy}(i_1', p_1, i_2, p_2)\mathit{SS}\ \mathit{SS}' \wedge$
$\quad\quad \mathit{lbl} = \mathit{tid}\cdot\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2)$
$\quad \mathbf{else}$
$\quad\quad \mathit{st}' = \mathit{SYN\_SENT} \wedge \mathit{is}_2' = \uparrow i_2 \wedge \mathit{ps}_2' = \uparrow p_2 \wedge$
$\quad\quad\quad \mathit{lbl} = \mathit{tid}\cdot\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2) \wedge$
$\quad\quad\quad \mathit{SS} = \mathit{SS}_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s)] \wedge$
$\quad\quad\quad \mathrm{write}(i_1', p_1', i_2, p_2)(\mathit{oflgs}, \mathit{odata})s\ s' \wedge$
$\quad\quad\quad \mathit{SS}' = \mathit{SS}_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1', p_1', i_2, p_2), s')])$

)

Description

From thread *tid*, a $\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2)$ 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 $\mathit{tid}\cdot\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2)$ transition is made, leaving the thread state *Connect2*(*sid*). If the socket is non-blocking (its *O_NONBLOCK* flag is set) and the segment enqueuing was successful then the call will fail: a $\mathit{tid}\cdot\mathit{connect}(\mathit{fd}, i_2, \uparrow p_2)$ 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ 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 *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 *connect*() call is made.
WinXP	If there is an early buffer allocation failure when enqueuing the segment, then it will not be placed on the host's out-queue and $es' = ENOBUFS$; the socket's control block will be cb' with its *snd_nxt* and *snd_max* fields set to the intial sequence number, its *last_ack_seen* and *rcv_adv* fields set to 0, its *tt_delack* option set to $*$, its *tt_rexmt* 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 case 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**

$$(h \, \langle\!| ts := ts \oplus (tid \mapsto (Connect2\ sid)_d) |\!\rangle, SS, MM)$$
$$\xrightarrow{\tau} (h \, \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}) |\!\rangle, SS, MM)$$

$$\begin{aligned}
&\text{TCP_PROTO}(tcp_sock) = (h.socks[sid]).pr \wedge \\
&tcp_sock.st \in \{ESTABLISHED; CLOSE_WAIT\} \wedge \\
&(\neg \exists tid'\ d'.(tid' \in \mathbf{dom}(ts)) \wedge (tid' \neq tid) \wedge \\
&\qquad ts[tid'] = (Connect2\ sid)_{d'})
\end{aligned}$$

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**

$$\begin{array}{lcl}
(h \, \langle\!| ts := ts \oplus (tid \mapsto (Connect2\ sid)_d); & \xrightarrow{\tau} & (h \, \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched_timer}); \\
socks := socks \oplus & & socks := socks \oplus \\
\quad [(sid, sock \, \langle\!| es := \uparrow e |\!\rangle)] |\!\rangle, & & \quad [(sid, sock \, \langle\!| es := * |\!\rangle)] |\!\rangle, \\
SS, MM) & & SS, MM)
\end{array}$$

$$\begin{aligned}
&\text{TCP_PROTO}(tcp_sock) = sock.pr \wedge \\
&tcp_sock.st = CLOSED
\end{aligned}$$

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 *bsd_cantconnect* flag in the control block would have been set, the socket's *cantsndmore* and *cantrcvmore* flags would have been set and its local address binding would have been removed. This renders the sockets useless: call to *bind*(), *connect*(), and *listen*() will all fail.

connect_4 **tcp: slow urgent fail** **Fail: socket has pending error**

$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Connect2\ sid)_d);$ $\xrightarrow{\tau}$ $(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer});$
$socks := socks \oplus$ $\quad$ $socks := socks \oplus$
$[(sid, sock)]\rangle\!\rangle,$ $\quad$ $[(sid, sock')]\rangle\!\rangle,$
$SS, MM)$ $\quad$ $SS', MM)$

$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, \uparrow err, \mathbf{F}, \mathbf{F},$
$\quad \text{TCP\_Sock}(SYN\_SENT, cb, *)) \wedge$

(* 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. *)

$\neg(windows\_arch\ h.arch \wedge err = EINVAL) \wedge$
if $bsd\_arch\ h.arch$ **then**
  if $(err = EADDRNOTAVAIL)$ **then**

    $sock' = \text{Sock}(\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \mathbf{F},$
      $\text{TCP\_Sock}(SYN\_SENT, cb, *)) \wedge$
    $SS' = SS$
  else
    $sock' = \text{Sock}(\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F},$
      $\text{TCP\_Sock}(CLOSED, \text{initial\_cb}, *)) \wedge$
    case ps_1 **of** $\uparrow p_1 \rightarrow \text{destroy}(i_1, p_1, i_2, p_2) SS\ SS'$
      $\parallel * \rightarrow SS' = SS$
else
  (* close the socket, but do not shutdown for reading/writing *)
  $sock' = \text{Sock}(\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F},$
    $\text{TCP\_Sock}(CLOSED, cb', *)) \wedge$
  $cb' = \text{initial\_cb} \wedge$
  case ps_1 **of** $\uparrow p_1 \rightarrow \text{destroy}(i_1, p_1, i_2, p_2) SS\ SS'$
    $\parallel * \rightarrow SS' = SS$

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 *EADDRNOTAVAIL* then the error is cleared and returned but the rest of the socket stays the same: it is in state *SYN_SENT* so the *SYN* will be retransmitted until it times out. If the pending error is not *EADDRNOTAVAIL* then the socket is reset as above except that the the socket's local ip and port are cleared
WinXP	If the error is *EINVAL* then this rule does not apply.

connect_4a **tcp: fast fail** **Fail with pending error**

$$
\begin{aligned}
&(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\
&\quad socks := socks \oplus\\
&\quad\quad [(sid, sock\ \langle\!|es := \uparrow err|\!\rangle)]|\!\rangle,\\
&\quad SS, MM)\\
&\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } err))_{sched\_timer});\\
&\quad socks := socks \oplus\\
&\quad\quad [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle,\\
&\quad SS, MM)
\end{aligned}
$$

$$
\begin{aligned}
&fd \in \mathbf{dom}(h.fds) \wedge\\
&fid = h.fds[fd] \wedge\\
&h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge\\
&\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge\\
&tcp\_sock.st \in \{CLOSED\}
\end{aligned}
$$

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 $tid \cdot connect(fd, ip, port)$ 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**

$$
\begin{aligned}
&(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)\\
&\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } err))_{sched\_timer})|\!\rangle, SS, MM)
\end{aligned}
$$

$fd \in \mathbf{dom}(h.fds) \wedge$

$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
case $tcp\_sock.st$ **of**
$SYN\_SENT \rightarrow$ **if** $ff.b(O\_NONBLOCK) = \mathbf{T}$ **then** $err = EALREADY$ (* 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 *)
$SYN\_RECEIVED \rightarrow$ **if** $ff.b(O\_NONBLOCK) = \mathbf{T}$ **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 *)
$LISTEN \rightarrow$ **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 *)
$ESTABLISHED \rightarrow err = EISCONN$ ‖ (* socket already connected *)
$FIN\_WAIT\_1 \rightarrow err = EISCONN$ ‖ (* socket already connected *)
$FIN\_WAIT\_2 \rightarrow err = EISCONN$ ‖ (* socket already connected *)
$CLOSING \rightarrow err = EISCONN$ ‖ (* socket already connected *)
$CLOSE\_WAIT \rightarrow err = EISCONN$ ‖ (* socket already connected *)
$LAST\_ACK \rightarrow err = EISCONN$ ‖ (* socket already connected; seems that fd is valid in this state *)
$TIME\_WAIT \rightarrow (windows\_arch\ h.arch \vee linux\_arch\ h.arch) \wedge err = EISCONN$ ‖
(* BSD allows a $TIME\_WAIT$ socket to be reconnected *)
$CLOSED \rightarrow err = EINVAL \wedge bsd\_arch\ h.arch$

Description

From thread *tid*, which is in the *Run* state, a $connect(fd, i_2, \uparrow p_2)$ 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 $err = EALREADY$ (*EISCONN* on FreeBSD); if it is in state *LISTEN* then on WinXP $err = EINVAL$, on FreeBSD $err = EOPNOTSUPP$, and on Linux $err = EISCONN$; if it is in state *ESTABLISHED*, *FIN_WAIT_1*, *FIN_WAIT_2*, *CLOSING*, *CLOSE_WAIT*, or *TIME_WAIT* on Linux and WinXP, $err = EISCONN$; if it is in state *CLOSED* on FreeBSD and has its *bsd_cantconnect* flag set then $err = EINVAL$.

A $tid \cdot connect(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ err)$.

Variations

FreeBSD	If the socket is in state *TIME_WAIT* then the call does not fail: the socket may be reconnected by *connect_1* (p84).

connect_5a **all: fast fail** **Fail: no route to host**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|is_1 := *; ps_1 := ps_1|\!\rangle)]|\!\rangle,$
$SS, MM)$

$$\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} \begin{array}{l}(h \ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } err))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, sock \ \langle\!|is_1 := is_1'; ps_1 := ps_1'|\!\rangle)]; \\ bound := bound|\!\rangle, \\ SS, MM)\end{array}$$

(* REMARK although this rule may result in a quad becoming bound, we assume (i_2, p_2) not bound *)
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$(\mathbf{if}\ bsd\_arch\ h.arch \wedge \text{proto\_of}\ sock.pr = PROTO\_TCP\ \mathbf{then}$
$\quad is_1' = \uparrow i_1' \wedge i_1' \in local\_primary\_ips\ h.ifds \wedge$
$\quad ps_1' = \uparrow p_1' \wedge p_1' \in \text{autobind}(ps_1, PROTO\_TCP, h, h.socks) \wedge$
$\quad (\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h.bound\ \mathbf{else}\ bound = h.bound)$
$\mathbf{else}\ is_1' = * \wedge ps_1' = ps_1 \wedge bound = h.bound) \wedge$
$\mathbf{case}\ test\_outroute\_ip(i_2, h.rttab, h.ifds, h.arch)\ \mathbf{of}$
$\quad \uparrow e \rightarrow err = e$
$\|\ \_other29 \rightarrow \mathbf{F} \wedge$
$(\text{proto\_of}\ sock.pr = PROTO\_UDP \implies \neg bsd\_arch\ h.arch)$

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 socket identified by *sid* which does not have a local IP address set. The *test_outroute_ip* 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ 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 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**

$$\begin{array}{l}(h \ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d); \\ socks := socks \oplus \\ \quad [(sid, sock)]; \\ bound := bound|\!\rangle, \\ SS, MM)\end{array}$$

$$\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} \begin{array}{l}(h \ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EADDRINUSE))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, sock \ \langle\!|is_1 := is_1'; ps_1 := \uparrow p_1'; is_2 := is_2'; ps_2 := ps_2'|\!\rangle)]; \\ bound := bound'|\!\rangle, \\ SS, MM)\end{array}$$

$fd \in \mathbf{dom}(h.fds) \wedge$

$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$
$i_1' \in auto\_outroute(i_2, sock.is_1, h.rttab, h.ifds) \wedge$
$p_1' \in \text{autobind}(sock.ps_1, (\text{proto\_of } sock.pr), h, h.socks) \wedge$
$(\textbf{if } sock.ps_1 = * \textbf{ then } bound' = sid :: bound \textbf{ else } bound' = bound) \wedge$
$(\text{proto\_of } sock.pr = PROTO\_UDP \implies \neg(linux\_arch\ h.arch \vee windows\_arch\ h.arch)) \wedge$
$(\exists(sid', s) :: socks \backslash\backslash sid.$
$\quad s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge$
$\quad s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge$
$\quad \text{proto\_eq } sock.pr\ s.pr) \wedge$
$(\textbf{if } \text{proto\_of } sock.pr = PROTO\_UDP \textbf{ then}$
$\quad \textbf{if } sock.is_2 = * \textbf{ then } is_1' = sock.is_1 \wedge is_2' = * \wedge ps_2' = *$
$\quad \textbf{else } is_1' = * \wedge is_2' = * \wedge ps_2' = *$
$\textbf{else } is_1' = sock.is_1 \wedge is_2' = sock.is_2 \wedge ps_2' = sock.ps_2)$

Description

From thread *tid*, which is in the *Run* state, a $connect(fd, i_2, \uparrow p_2)$ 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $Ret(\text{FAIL } EADDRINUSE)$: 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 **<u>all: fast fail</u>** **Fail with** ***EADDRNOTAVAIL*****: no ephemeral ports left**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EADDRNOTAVAIL))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$
$(h.socks[sid]).ps_1 = * \wedge$
$\text{autobind}(*, (\text{proto\_of}(h.socks[sid]).pr), h, h.socks) = \emptyset$

Description

From thread *tid*, which is in the *Run* state, a $connect(fd, i_2, \uparrow p_2)$ 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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

A $tid \cdot connect(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $Ret(\mathrm{FAIL}\ EADDRNOTAVAIL)$.

connect_5d **tcp: block** **Block, entering state** *Connect2*: **connection attempt already in progress and connect called with blocking semantics**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Connect2(sid))_{never\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$\mathrm{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
$ff.b(O\_NONBLOCK) = \mathbf{F} \wedge$
$linux\_arch\ h.arch \wedge$
$tcp\_sock.st \in \{SYN\_SENT; SYN\_RECEIVED\}$

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 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 $tid \cdot connect(fd, i_2, \uparrow p_2)$ 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**

$$\begin{aligned}&(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ &socks := socks \oplus\\ &\quad [(sid, sock\ \langle\!|cantsndmore := \mathbf{T}; pr := \mathrm{TCP\_PROTO}(tcp\ \langle\!|st := CLOSED|\!\rangle)|\!\rangle)]|\!\rangle,\\ &SS, MM)\end{aligned}$$
$$\xrightarrow{tid \cdot connect(fd, i_2, \uparrow p_2)}$$
$$\begin{aligned}&(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EINVAL))_{sched\_timer});\\ &socks := socks \oplus\\ &\quad [(sid, sock\ \langle\!|cantsndmore := \mathbf{T}; pr := \mathrm{TCP\_PROTO}(tcp\ \langle\!|st := CLOSED|\!\rangle)|\!\rangle)]|\!\rangle,\\ &SS, MM)\end{aligned}$$

$bsd\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff)$

Description

On FreeBSD, 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 which is in state $CLOSED$ and has been shutdown for writing.

A $tid \cdot connect(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state $Ret(\text{FAIL } EINVAL)$.

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** $*, ps_1, *, *$

$$
\begin{array}{l}
(h_0, SS, MM) \\
\xrightarrow{tid \cdot connect(fd, i_2, ps_2)} \\
\quad (h_0 \ \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}); \\
\quad socks := socks \oplus \\
\quad\quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1', \uparrow p_1', \uparrow i_2, ps_2, es, cantsndmore', cantrcvmore, \text{UDP\_PROTO}(udp)))]; \\
\quad bound := bound \\
\quad |\!\rangle, \\
\quad SS, MM)
\end{array}
$$

$$
\begin{array}{l}
h_0 = h \ \langle\!| \ ts := ts \oplus (tid \mapsto (Run)_d); \\
\qquad socks := socks \oplus \\
\qquad\quad [(sid, \text{SOCK}(\uparrow fid, sf, *, ps_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))] \\
\qquad |\!\rangle \wedge \\
fd \ \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h_0.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge \\
p_1' \ \in \ \text{autobind}(ps_1, PROTO\_UDP, h_0, h_0.socks) \wedge \\
(\mathbf{if} \ ps_1 = * \ \mathbf{then} \ bound = sid :: h_0.bound \ \mathbf{else} \ bound = h_0.bound) \wedge \\
i_1' \ \in \ auto\_outroute(i_2, *, h_0.rttab, h_0.ifds) \wedge \\
\neg(\exists (sid', s) :: (h_0.socks \backslash\backslash sid). \\
\qquad s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge \\
\qquad s.is_2 = \uparrow i_2 \wedge s.ps_2 = ps_2 \wedge \\
\qquad \text{proto\_of} \ s.pr = PROTO\_UDP \wedge \\
\qquad bsd\_arch \ h.arch) \wedge \\
(bsd\_arch \ h.arch \implies ps_2 \neq * \wedge es = *) \wedge \\
(\mathbf{if} \ windows\_arch \ h.arch \ \mathbf{then} \ cantsndmore' = \mathbf{F} \\
\ \mathbf{else} \ cantsndmore' = cantsndmore)
\end{array}
$$

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 $connect(fd, i_2, ps_2)$ 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 $tid \cdot connect(fd, i_2, ps_2)$ transition is made, leaving the thread state $Ret(OK())$. 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: $bound = sid :: h_0.bound$.

Variations

FreeBSD	As above, with the additional conditions that a foreign port is specified in the *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 *connect*() call has been made.

connect_8 **udp: fast succeed** **Set peer address on socket with local address set**

$$(h_0, SS, MM)$$

$$\xrightarrow{tid \cdot connect(fd, i, ps)}$$

$$\begin{array}{l}(h \; \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i, ps, es, cantsndmore', cantrcvmore, \text{UDP\_PROTO}(udp)))]\rangle\!\rangle, \\ SS, MM)\end{array}$$

$$\begin{array}{l}h_0 = h \; \langle\!\langle \; ts := ts \oplus (tid \mapsto (Run)_d); \\ \qquad socks := socks \oplus \\ \qquad\quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]\rangle\!\rangle \wedge \\ fd \; \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge \\ (bsd\_arch \; h.arch \implies ps \neq * \wedge es = *) \wedge \\ (\mathbf{if} \; windows\_arch \; h.arch \; \mathbf{then} \; cantsndmore' = \mathbf{F} \\ \; \mathbf{else} \; cantsndmore' = cantsndmore) \wedge \\ \neg(\exists (sid', s) :: (h_0.socks \backslash\backslash sid). \\ \qquad s.is_1 = \uparrow i_1 \wedge s.ps_1 = \uparrow p_1 \wedge \\ \qquad s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge \\ \qquad \text{proto\_of} \; s.pr = PROTO\_UDP \wedge \\ \qquad bsd\_arch \; h.arch)\end{array}$$

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 *connect*(fd, i, ps) call is made.

The call succeeds: a $tid \cdot connect(fd, i, ps)$ transition is made, leaving the thread in state $Ret(OK())$. 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 *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 $(\uparrow i_1', \uparrow p_1', \uparrow i, ps)$.
WinXP	As above, with the additional effect that if the socket was shutdown for writing when the *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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\ SS, MM)$$

$$\xrightarrow{tid \cdot connect(fd, i, *)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EADDRNOTAVAIL))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|is_1 := is_1; is_2 := *; ps_2 := *; pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\ SS, MM)$$

$bsd\_arch\ h.arch\ \wedge$
$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$(\mathbf{if}\ sock.is_2 \neq *\ \mathbf{then}\ is_1 = *\ \mathbf{else}\ is_1 = sock.is_1)$

Description

On FreeBSD, consider a UDP socket sid referenced by fd. From thread tid, which is in the Run state, a $connect(fd, i, *)$ call is made. Because no port is specified, the call fails with an $EADDRNOTAVAIL$ error.

A $tid \cdot connect(fd, i, *)$ transition is made, leaving the thread state $Ret(\mathrm{FAIL}\ EADDRNOTAVAIL)$. 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**

$$(h_0, SS, MM) \xrightarrow{tid \cdot connect(fd, i, ps)} (h_0\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ err))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|is_2 := \uparrow i; ps_2 := ps; es := *; pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\ SS, MM)$$

$bsd\_arch\ h.arch\ \wedge$
$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|\ es := \uparrow err; pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle \wedge$
$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$ps \neq * \wedge$
$\neg(\exists (sid', s) :: (h_0.socks \backslash\backslash sid).$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$s.is_1 = sock.is_1 \wedge s.ps_1 = sock.ps_1 \wedge$
$s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge$
$\text{proto\_of }\ s.pr = PROTO\_UDP)$

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 $connect(fd, i, ps)$ 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 $tid \cdot connect(fd, i, ps)$ transition is made, leaving the thread state *Ret*(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.

7.5 *disconnect*() (TCP and UDP)

$disconnect : fd \rightarrow$ unit

A call to $disconnect(fd)$, 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 *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* (p74)).

7.5.1 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 `AF_UNSPEC` is not supported. This can be the result for a successful *disconnect*() 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 *fd* 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.

7.5.2 Common cases

disconnect_1; *return_1*

7.5.3 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.

7.5.4 Summary

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
disconnect_5	**tcp: fast fail**	Succeed on Linux, possibly dropping the connection
disconnect_1	**udp: fast succeed**	Unset socket's peer address
disconnect_2	**udp: fast succeed**	Unset socket's peer address and autobind local port
disconnect_3	**udp: fast fail**	Fail with *EAGAIN*, *EADDRNOTAVAIL*, or *ENOBUFS*: there are no ephemeral ports left

7.5.5 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$

$$\xrightarrow{tid\cdot disconnect(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
$\neg(linux\_arch\ h.arch) \wedge$
case *tcp_sock.st* **of**
  CLOSED → **if** *bsd_arch h.arch* **then** $err = EINVAL \vee err = EAFNOSUPPORT$
    else $err = EAFNOSUPPORT$ ‖
  LISTEN → **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 → $err = EALREADY$ ‖ (* connection already in progress *)
  SYN_RECEIVED → $err = EALREADY$ ‖ (* connection already in progress *)
  ESTABLISHED → $err = EISCONN$ ‖ (* socket already connected *)
  TIME_WAIT → **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 → $err = EISCONN$ (* 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 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**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus [(sid, sock)];$
$oq := oq|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot disconnect(fd)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});$
$socks := socks \oplus [(sid, sock')];$
$oq := oq'|\!\rangle,$
$SS', MM)$

$linux\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
(**if** $tcp\_sock.st \in \{SYN\_RECEIVED; ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSE\_WAIT\}$ **then**
  $\text{tcp\_drop\_and\_close}\ h.arch * sock(sock', (oflgs, odata)) \wedge$
  if exists_quad_of *sock* **then**
    $\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(\text{quad\_of}\ sock), s)] \wedge$
    $\text{write}(\text{quad\_of}\ sock)(oflgs, odata)s\ s' \wedge$
    $\text{destroy}(\text{quad\_of}\ sock)(S_0 \oplus [(\text{streamid\_of\_quad}(\text{quad\_of}\ sock), s')])SS'$
  else

$SS' = SS$
else
$sock = sock' \wedge$
$oq = oq' \wedge$
$SS' = SS$
)

Description

On Linux, consider a TCP socket *sid*, referenced by *fd*. From thread *tid*, which is in the *Run* state, a *disconnect*(*fd*) call is made and succeeds.

A *tid*·*disconnect*(*fd*) transition is made, leaving the thread state *Ret*(*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 *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**

$(h\ \langle\!|ts := ts\_ \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]$
$|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot disconnect(fd)}$

$(h\ \langle\!|ts := ts\_ \oplus (tid \mapsto (Ret(ret))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, *, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]$
$|\!\rangle,$
$SS, MM)$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
(**if** $linux\_arch\ h.arch$ **then** $ret = OK()$
else if $windows\_arch\ h.arch \wedge \exists i_2'.is_2 = \uparrow i_2'$ **then** $ret = OK()$
else $ret = \text{FAIL}\ EAFNOSUPPORT$)

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 *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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

A $tid \cdot disconnect(fd)$ transition is made, leaving the thread state $Ret(OK())$ or $Ret(\text{FAIL } EAFNOSUPPORT)$. 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**

$$(h_0, SS, MM)$$

$$\xrightarrow{tid \cdot disconnect\ fd}$$

$$\begin{array}{l} (h_0\ \langle\!|ts := ts\_ \oplus (tid \mapsto (Ret(ret))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, *, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]; \\ bound := sid :: h_0.bound|\!\rangle, \\ SS, MM) \end{array}$$

$$\begin{array}{l} h_0 = h\ \langle\!|\ ts := ts\_ \oplus (tid \mapsto (Run)_d); \\ \quad socks := socks \oplus \\ \quad\quad [(sid, \textsc{Sock}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle \wedge \\ fd\ \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge \\ p_1\ \in\ \text{autobind}(*, PROTO\_UDP, h_0, h_0.socks) \wedge \\ (\mathbf{if}\ \ linux\_arch\ h.arch\ \ \mathbf{then}\ \ ret = OK() \\ \ \mathbf{else}\ \ ret = (\text{FAIL } EAFNOSUPPORT)) \end{array}$$

Description

Consider a UDP socket sid referenced by fd and with binding quad $(*, *, *, *)$. From thread tid, which is in the Run state, a $disconnect(fd)$ call is made. The call succeeds on Linux and fails with an *EAFNOSUPPORT* error on FreeBSD and WinXP.

A $tid \cdot disconnect(fd)$ transition is made, leaving the thread either in state $Ret(OK())$, or in state $Ret(\text{FAIL } EAFNOSUPPORT)$. The socket is autobound to a local ephemeral port p_1', 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**

$$(h_0, SS, MM) \xrightarrow{tid \cdot disconnect\ fd} (h_0\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer}) |\!\rangle, SS, MM)$$

$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);$
$\quad socks := socks \oplus$
$\quad\quad [(sid, \text{SOCK}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))] |\!\rangle \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{autobind}(*, PROTO\_UDP, h_0, h_0.socks) = \emptyset \wedge$
$e \in \{EAGAIN; EADDRNOTAVAIL; ENOBUFS\}$

Description

Consider a UDP socket *sid* referenced by *fd* and with binding quad $*, *, *, *$. From thread *tid*, which is in the *Run* state, a *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*·*disconnect*(*fd*) transition is made, leaving the thread state *Ret*(FAIL *e*) where *e* is one of the above errors.

7.6 *dup*() (TCP and UDP)

$dup : fd \rightarrow fd$

A call to *dup*(*fd*) creates and returns a new file descriptor referring to the open file description referred to by the file descriptor *fd*. A successful *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.

7.6.1 Errors

A call to *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.

7.6.2 Common cases

dup_1; *return_1*

7.6.3 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.

7.6.4 Summary

dup_1	**all: fast succeed**	Successfully duplicate file descriptor
dup_2	**all: fast fail**	Fail with *EMFILE*: no more file descriptors available

7.6.5 Rules

dup_1 **all: fast succeed** **Successfully duplicate file descriptor**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d);\ fds := fds \rangle\!\rangle,\ SS, MM) \xrightarrow{tid \cdot dup(fd)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK\ fd'))_{sched\_timer});\ fds := fds' \rangle\!\rangle,\ SS, MM)$$

$unix\_arch\ h.arch\ \wedge$
$fd\ \in \mathbf{dom}(fds)\ \wedge$
$fid = fds[fd]\ \wedge$
$nextfd\ h.arch\ fds\ fd'\ \wedge$
$fd' < OPEN\_MAX\_FD\ \wedge$
$fds' = fds \oplus (fd', fid)$

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* function. fd' is less than the maximum open file descriptor, $OPEN\_MAX\_FD$. The call succeeds returning fd'.

A $tid \cdot dup(fd)$ 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 *dup*() call on WinXP.

dup_2 **all: fast fail** **Fail with *EMFILE*: no more file descriptors available**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d) \rangle\!\rangle, SS, MM) \xrightarrow{tid \cdot dup(fd)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EMFILE))_{sched\_timer}) \rangle\!\rangle, SS, MM)$$

$unix\_arch\ h.arch\ \wedge$
$fd\ \in \mathbf{dom}(h.fds)\ \wedge$
$(\mathbf{card}(\mathbf{dom}(h.fds)) + 1) \geq OPEN\_MAX$

Description

From thread *tid*, which is in the *Run* state, a $dup(fd)$ 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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

A $tid \cdot dup(fd)$ transition is made, leaving the thread state Ret(FAIL $EMFILE$).

Variations

WinXP	This rule does not apply: there is no *dup*() call on WinXP.

7.7 *dupfd*() (TCP and UDP)

$dupfd : fd * \mathsf{int} \rightarrow fd$

A call to $dupfd(fd, n)$ creates and returns a new file desciptor referring to the open file description referred to by the file descriptor fd.

A successful *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.

7.7.1 Errors

A call to *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.

7.7.2 Common cases

dupfd_1; *return_1*

7.7.3 API

dupfd() is Posix *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);`

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 *dupfd*().
- `cmd` is the command to run on the specified file descriptor. For the model *dupfd*() this command is set to `F_DUPFD`.
- 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 was successful then the returned `int` is the new file descriptor.

The FreeBSD and Linux interfaces are similar. This call does not exist on WinXP.

7.7.4 Model details

Note that *dupfd*() is `fcntl()` with `F_DUPFD` rather than the similar but different `dup2()`.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

7.7.5 Summary

dupfd_1	**all: fast succeed**	Successfully create a duplicate file descriptor greater than or equal to n
dupfd_3	**all: fast fail**	Fail with *EINVAL*: n is negative or greater than the maximum allowed file descriptor
dupfd_4	**all: fast fail**	Fail with *EMFILE*: no more file descriptors available

7.7.6 Rules

dupfd_1 **all: fast succeed** **Successfully create a duplicate file descriptor greater than or equal to** n

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d); fds := fds|\!\rangle, SS, MM) \xrightarrow{tid \cdot dupfd(fd, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK\ fd'))_{sched\_timer}); fds := fds'|\!\rangle, SS, MM)$$

$unix\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$n \geq 0 \wedge$
$FD(\mathbf{num}\ n) < OPEN\_MAX\_FD \wedge$
$fd' = FD(\mathbf{least}\ n'.\ \mathbf{num}\ n \leq n' \wedge FD\ n' < OPEN\_MAX\_FD \wedge FD\ n' \notin \mathbf{dom}(fds)) \wedge$
$fds' = fds \oplus (fd', fid)$

Description

From thread *tid*, which is in the *Run* state, a $dupfd(fd, n)$ call is made. The host's finite map of file descriptors is *fds*, and *fd* is a valid file descriptor in *fds*, referring to an open file description identified by *fid*. n is non-negative. A file descriptor fd' can be created, where it is the least free file descriptor greater than or equal to n, and less than the maximum allowed file descriptor, *OPEN_MAX_FD*. The call succeeds, returning this new file descriptor fd'.

A $tid \cdot dupfd(fd, n)$ transition is made, leaving the thread state $Ret(OKfd')$. An entry mapping fd' to the open file description *fid* is added to *fds*, resulting in a new finite map of file descriptors for the host, fds'.

Variations

WinXP	This rule does not apply: there is no *dupfd*() call on WinXP.

dupfd_3 **all: fast fail** **Fail with** *EINVAL*: n **is negative or greater than the maximum allowed file descriptor**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid \cdot dupfd(fd, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ err))_{sched\_timer})|\!\rangle, SS, MM)$$

$unix\_arch\ h.arch \wedge$
$n < 0 \vee \mathbf{num}\ n \geq OPEN\_MAX \wedge$
$err = (\mathbf{if}\ bsd\_arch\ h.arch\ \mathbf{then}\ EBADF\ \mathbf{else}\ EINVAL)$

Description

From thread *tid*, which is in the *Run* state, a $dupfd(fd, n)$ 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 $tid \cdot dupfd(fd, n)$ 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 <u>**all: fast fail**</u> **Fail with *EMFILE*: no more file descriptors available**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot dupfd(fd, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EMFILE))_{sched\_timer})|\!\rangle, SS, MM)$$

$unix\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$n \geq 0 \wedge$
$fd' = FD(\mathbf{least}\ n'.\ \mathbf{num}\ n \leq n' \wedge OPEN\_MAX\_FD \leq FD\ n' \wedge FD\ n' \notin \mathbf{dom}(h.fds))$

Description

From thread *tid*, which is in the *Run* state, a $dupfd(fd, n)$ 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 $tid \cdot dupfd(fd, n)$ transition is made, leaving the thread state *Ret*(FAIL *EMFILE*).

Variations

WinXP	This rule does not apply: there is no *dupfd*() call on WinXP.

7.8 *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.

7.8.1 Errors

A call to *getfileflags*() can fail with the error below, in which case the corresponding exception is raised:

EBADF	The file descriptor passed is not a valid file descriptor.

7.8.2 Common cases

A call to *getfileflags*() is made, returning the flags set: *getfileflags_1*; *return_1*

7.8.3 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()`.

7.8.4 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.

7.8.5 Summary

getfileflags_1	**all: fast succeed**	Return list of file flags currently set for an open file description

7.8.6 Rules

getfileflags_1 **<u>all: fast succeed</u>** **Return list of file flags currently set for an open file description**

$$(h\ \langle\!| ts := ts \oplus (tid \mapsto (Run)_d) |\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot getfileflags(fd)} \quad (h\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK\ flags))_{sched\_timer}) |\!\rangle, SS, MM)$$

$fd \ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(ft, ff) \wedge$
$flags \ \in \ ORDERINGS\ ff.b$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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*)).

7.9 *getifaddrs*() (TCP and UDP)

getifaddrs : unit → (*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*().

7.9.1 Errors

EINTR	The system was interrupted by a caught signal.
EBADF	The file descriptor passed is not a valid file descriptor.

7.9.2 Common cases

getifaddrs_1; *return_1*

7.9.3 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*().

7.9.4 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.

7.9.5 Summary

getifaddrs_1	**all: fast succeed**	Successfully return host interface information

7.9.6 Rules

getifaddrs_1 **all: fast succeed** **Successfully return host interface information**

$$(h\ \ ts := ts \oplus (tid \mapsto (Run)_d), SS, MM)$$

$$\xrightarrow{tid \cdot getifaddrs()} (h\ \ ts := ts \oplus (tid \mapsto (Ret(OK\ \ iflist))_{sched\_timer}), SS, MM)$$

$$ifidlist \in ORDERINGS\ ifidset \wedge$$
$$\mathbf{length}\ ifidlist = \mathbf{length}\ iflist \wedge$$

$$ifidset = \{(ifid, hifd) \mid ifid \in \mathbf{dom}(h.ifds) \wedge hifd = h.ifds[ifid]\} \wedge$$

$$\mathbf{every}\ \mathbf{I}(\mathbf{map2}(\lambda(ifid, hifd)(ifid', primary, ipslist, netmask).(ifid' = ifid \wedge primary = hifd.primary \wedge ipslist \in ORDERINGS\ hifd.ipset \wedge netmask = hifd.netmask))\ ifidlist\ iflist)$$

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.

7.10 *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 $sock.ps_2 = \uparrow p_2$.

7.10.1 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.

7.10.2 Common cases

getpeername_1; *return_1*

7.10.3 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()`.

7.10.4 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.

7.10.5 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

7.10.6 Rules

getpeername_1 **all: fast succeed** **Successfully return socket's peer address**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot getpeername(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(i_2, p_2)))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = h.socks[sid] \wedge$
$sock.is_2 = \uparrow i_2 \wedge$
$(sock.ps_2 = \uparrow p_2 \vee (windows\_arch\ h.arch \wedge sock.ps_2 = * \wedge$
$(p_2 = Port\ 0) \wedge \text{proto\_of}\ sock.pr = PROTO\_UDP)) \wedge$
$((\forall tcp\_sock.sock.pr = \text{TCP\_PROTO}(tcp\_sock) \implies$
$tcp\_sock.st \in \{ESTABLISHED; CLOSE\_WAIT; LAST\_ACK;$
$FIN\_WAIT\_1; CLOSING\} \vee$
$(\neg sock.cantrcvmore \wedge tcp\_sock.st = FIN\_WAIT\_2) \vee$
$(linux\_arch\ h.arch \wedge tcp\_sock.st = SYN\_RECEIVED) \vee$
(* BSD listen bug *)
$(bsd\_arch\ h.arch \wedge tcp\_sock.st = LISTEN)) \vee$
$windows\_arch\ h.arch)$

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 $Ret(OK(i_2, p_2))$.

Variations

FreeBSD	If *sock* is a TCP socket then it may be in state *LISTEN*; this is due to the FreeBSD bug that allows *listen*() to be called on a synchronised socket.
Linux	If *sock* is a TCP socket then it may also be in state *SYN_RECEIVED*.
WinXP	If *sock* is a UDP socket and has no peer port set, $sock.ps_2 = *$ then the call may still succeed with $p_2 = Port\ 0$. Additionally, if *sock* 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$\xrightarrow{tid \cdot getpeername(fd)}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ ENOTCONN))_{sched\_timer})|\!\rangle, SS, MM)$

$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = h.socks[sid] \wedge$
$\neg(sock.is_2 \neq * \wedge$
$\quad (sock.ps_2 \neq * \vee (windows\_arch\ h.arch \wedge \mathrm{proto\_of}\ \ sock.pr = PROTO\_UDP)) \wedge$
$\quad (\forall tcp\_sock.sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \implies$
$\qquad tcp\_sock.st\ \in \{ESTABLISHED; CLOSE\_WAIT; LAST\_ACK; FIN\_WAIT\_1; CLOSING\} \vee$
$\qquad (\neg sock.cantrcvmore \wedge tcp\_sock.st = FIN\_WAIT\_2) \vee$
$\qquad (linux\_arch\ h.arch \wedge tcp\_sock.st = SYN\_RECEIVED) \vee$

$\qquad windows\_arch\ h.arch))$

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 *sock* is a TCP socket then it is not in state *SYN_RECEIVED*.
WinXP	As above, except that if *sock* 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.

7.11 *getsockbopt*() (TCP and UDP)

$getsockbopt : (fd * sockbflag) \rightarrow \mathsf{bool}$

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.

7.11.1 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.

7.11.2 Common cases

getsockbopt_1; *return_1*

7.11.3 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_OOBINLINE*, 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()`.

7.11.4 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.

7.11.5 Summary

getsockbopt_1	**all: fast succeed**	Successfully retrieve value of boolean socket flag
getsockbopt_2	**udp: fast succeed**	Fail with *ENOPROTOOPT*: option not valid on WinXP UDP socket

7.11.6 Rules

getsockbopt_1 **all: fast succeed** **Successfully retrieve value of boolean socket flag**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid \cdot getsockbopt(fd, f)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(sf.b(f))))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sf = (h.socks[sid]).sf \wedge$
$(windows\_arch\ h.arch \wedge \text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP$
$\implies f \notin \{SO\_KEEPALIVE\})$

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 $tid \cdot getsockbopt(fd, f)$ 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 sid is a UDP socket, then f cannot be *SO_KEEPALIVE* or *SO_OOBINLINE*.

getsockbopt_2 **udp: fast succeed** **Fail with *ENOPROTOOPT*: option not valid on WinXP UDP socket**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ socks := socks \oplus\\ \quad [(sid, sock\ \langle\!|pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array} \xrightarrow{tid \cdot getsockbopt(fd, f)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, sock\ \langle\!|pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array}$$

$windows\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$f \in \{SO\_KEEPALIVE\}$

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.

7.12 *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.

7.12.1 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:

EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

7.12.2 Common cases

getsockerr_1; *return_1*

getsockerr_2; *return_1*

7.12.3 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()`.

7.12.4 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.

7.12.5 Summary

getsockerr_1	**all: fast succeed**	Return successfully: no pending error
getsockerr_2	**all: fast fail**	Fail with pending error and clear the error

7.12.6 Rules

getsockerr_1 **all: fast succeed** **Return successfully: no pending error**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid\cdot getsockerr(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$(h.socks[sid]).es = *$

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 $tid\cdot getsockerr(fd)$ transition is made, leaving the thread state $Ret(OK())$.

getsockerr_2 **all: fast fail** **Fail with pending error and clear the error**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ socks := socks \oplus [(sid, sock)]|\!\rangle,\\ SS, MM)\end{array} \xrightarrow{tid\cdot getsockerr(fd)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } e))_{sched\_timer});\\ socks := socks \oplus [(sid, sock')]|\!\rangle,\\ SS, MM)\end{array}$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\uparrow e = sock.es \wedge$

$sock' = sock \llparenthesis es := * \rrparenthesis$

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 cleaing the error *e* from the socket.

7.13 *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.

7.13.1 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.

7.13.2 Common cases

getsocklistening_1; *return_1*

7.13.3 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.

7.13.4 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.

7.13.5 Summary

getsocklistening_1 **tcp: fast succeed**	Return successfully: **T** if socket is listening, **F** otherwise
getsocklistening_3 **tcp: fast fail**	Fail with *ENOPROTOOPT*: on FreeBSD operation not supported
getsocklistening_2 **udp: rc**	Return **F** or fail with *ENOPROTOOPT*: a UDP socket cannot be listening

7.13.6 Rules

getsocklistening_1 **tcp: fast succeed** **Return successfully: T if socket is listening, F otherwise**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot getsocklistening(fd)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK\ b))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)\ \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr\ \wedge$
$b = (tcp\_sock.st = LISTEN)\ \wedge$
$\neg(bsd\_arch\ h.arch)$

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 $tid \cdot getsocklistening(fd)$ 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 *getsocklistening_3*.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

getsocklistening_3 **tcp: fast fail** **Fail with** *ENOPROTOOPT***: on FreeBSD operation not supported**

$$(h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Run)_d)\rbrack\!\rangle, SS, MM) \xrightarrow{tid\cdot getsocklistening(fd)} (h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer})\rbrack\!\rangle, SS, MM)$$

$bsd\_arch\ h.arch\ \wedge$
$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)\ \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr$

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 *getsocklistening_1*.
WinXP	This rule does not apply: see *getsocklistening_1*.

getsocklistening_2 **udp: rc** **Return F or fail with** *ENOPROTOOPT***: a UDP socket cannot be listening**

$$(h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Run)_d)\rbrack\!\rangle, SS, MM) \xrightarrow{tid\cdot getsocklistening(fd)} (h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (Ret(ret))_{sched\_timer})\rbrack\!\rangle, SS, MM)$$

$\text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP\ \wedge$
$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)\ \wedge$
if $linux\_arch\ h.arch$ **then** $rc = fast\ succeed \wedge ret = OK\ \mathbf{F}$
else $rc = fast\ fail \wedge ret = \text{FAIL}\ ENOPROTOOPT$

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.

7.14 *getsockname*() (TCP and UDP)

getsockname : *fd* → (*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 successfull call will be ($sock.is_1$, $sock.ps_1$).

7.14.1 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 *cb.bsd_cantconnect* 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.

7.14.2 Common cases

getsockname_1; *return_1*

7.14.3 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↑*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 ↑ *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()`.

7.14.4 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.

7.14.5 Summary

getsockname_1	**all: fast succeed**	Successfully return socket's local address
getsockname_2	**tcp: fast fail**	Fail with *ECONNRESET*: previous connection attempt has failed on FreeBSD
getsockname_3	**all: fast fail**	Fail with *EINVAL*: socket not bound on WinXP

7.14.6 Rules

getsockname_1 **all: fast succeed** **Successfully return socket's local address**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d)\rangle\!\rangle, SS, MM)$$

$$\xrightarrow{tid\cdot getsockname(fd)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK(sock.is_1, sock.ps_1)))_{sched\_timer})\rangle\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = h.socks[sid] \wedge$
$(\mathbf{case}\ sock.pr\ \mathbf{of}$
$\quad \mathrm{TCP\_PROTO}(tcp\_sock) \rightarrow$
$\quad\quad bsd\_arch\ h.arch \implies \mathbf{T}\ \|$
$\quad \mathrm{UDP\_PROTO}(\_444) \rightarrow \mathbf{T}) \wedge$
$(windows\_arch\ h.arch \implies sock.is_1 \neq * \vee sock.ps_1 \neq *)$

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 $Ret(OK(sock.is_1, sock.ps_1))$.

Variations

FreeBSD	This rule does not apply if the socket's *bsd_cantconnect* flag is set in its control block and its local port is not set.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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**

$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus [(sid, sock)]\rangle\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot getsockname(fd)}$ $(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ ECONNRESET))_{sched\_timer});$
$socks := socks \oplus [(sid, sock)]\rangle\!\rangle,$
$SS, MM)$

$bsd\_arch\ h.arch\ \wedge$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock)\ \wedge$
$(sock.ps_1 = *)\ \wedge$

$fd\ \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff)$

Description

On FreeBSD, from thread *tid*, which is in the *Run* state, a *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*·*getsockname*(*fd*) transition is made, leaving the thread state *Ret*(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**

$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, sock\ \langle\!\langle is_1 := *; ps_1 := *\rangle\!\rangle)]\rangle\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot getsockname(fd)}$ $(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EINVAL))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!\langle is_1 := *; ps_1 := *\rangle\!\rangle)]\rangle\!\rangle,$
$SS, MM)$

$windows\_arch\ h.arch\ \wedge$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff)$

Description

On WinXP, a *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*·*getsockname*(*fd*) transition is made, leaving the thread state *Ret*(FAIL *EINVAL*).

Variations

Posix	This rule does not apply.
FreeBSD	This rule does not apply.
Linux	This rule does not apply.

7.15 *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*.

7.15.1 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.

7.15.2 Common cases

getsocknopt_1; *return_1*

7.15.3 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 `getsockopt()`. 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()`.

7.15.4 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.

7.15.5 Summary

getsocknopt_1	**all: fast succeed**	Successfully retrieve value of a numeric socket flag
getsocknopt_4	**all: fast fail**	Fail with *ENOPROTOOPT*: value of *SO_RCVLOWAT* and *SO_SNDLOWAT* not retrievable

7.15.6 Rules

getsocknopt_1 <u>**all: fast succeed**</u> **Successfully retrieve value of a numeric socket flag**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$

$$\xrightarrow{tid \cdot getsocknopt(fd, f)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(\mathbf{int\_of\_num}(sf.n(f)))))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$
$sf = (h.socks[sid]).sf \wedge$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$(windows\_arch\ h.arch \implies f \notin \{SO\_RCVLOWAT; SO\_SNDLOWAT\})$

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 $tid \cdot getsocknopt(fd, f)$ transition is made, leaving the thread state $Ret(OK(\mathbf{int\_of\_num}(sf.n(f))))$.

Variations

WinXP	The flag *f* is not *SO_RCVLOWAT* or *SO_SNDLOWAT*.

getsocknopt_4 **all: fast fail** **Fail with** *ENOPROTOOPT*: **value of** *SO_RCVLOWAT* **and** *SO_SNDLOWAT* **not retrievable**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot getsocknopt(fd, f)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer})|\!\rangle, SS, MM)$$

$windows\_arch\ h.arch\ \wedge$
$f \in \{SO\_RCVLOWAT; SO\_SNDLOWAT\}$

Description

From thread *tid*, which is in the *Run* state, a *getsocknopt*(*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 $tid \cdot getsocknopt(fd, f)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ ENOPROTOOPT)$.

Variations

FreeBSD	This rule does not apply.
Linux	This rule does not apply.

7.16 *getsocktopt*() (TCP and UDP)

$getsocktopt : (fd * socktflag) \rightarrow (\mathsf{int} * \mathsf{int})\ \mathsf{option}$

A call to *getsocktopt*(*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 *getsocktopt*() 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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

7.16.1 Errors

A call to *getsocktopt*() 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.

7.16.2 Common cases

getsocktopt_1; *return_1*

7.16.3 API

getsocktopt() 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 *getsocktopt*().
- `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 *getsocktopt*() 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 *getsocktopt*().
- 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()`.

7.16.4 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 *getsocktopt*().
- `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.

7.16.5 Summary

getsocktopt_1	**all: fast succeed**	Successfully retrieve value of time-option socket flag
getsocktopt_4	**all: fast fail**	Fail with *ENOPROTOOPT*: on WinXP *SO_LINGER* not retrievable for UDP sockets

7.16.6 Rules

getsocktopt_1 **all: fast succeed** **Successfully retrieve value of time-option socket flag**

$$(h \;\langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid \cdot getsocktopt(fd,f)} (h \;\langle\!|ts := ts \oplus (tid \mapsto (Ret(OK\;\; t))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sf = (h.socks[sid]).sf \wedge$
$t = tltimeopt\_of\_time(sf.t(f)) \wedge$
$\neg(windows\_arch\ h.arch \wedge \text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP \wedge$
$f = SO\_LINGER)$

Description

From thread *tid*, which is in the *Run* state, a *getsocktopt*(*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 $OK(t)$ where t is the value of the socket's flag f.

A *tid*·*getsocktopt*(*fd*, *f*) transition is made, leaving the thread state $Ret(OK\,t)$.

Model details

The return type is (int ∗ 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 *SO_LINGER*.

getsocktopt_4 **all: fast fail** **Fail with *ENOPROTOOPT*: on WinXP *SO_LINGER* not retrievable for UDP sockets**

$$(h \;\langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid \cdot getsocktopt(fd,f)} (h \;\langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer})|\!\rangle, SS, MM)$$

$windows\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP \wedge$
$f = SO\_LINGER$

Description

On WinXP, from thread *tid* which is in the *Run* state, a *getsocktopt*(*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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

A *tid*·*getsocktopt*(*fd*, *f*) transition is made, leaving the thread state *Ret*(*ENOPROTOOPT*).

Variations

FreeBSD	This rule does not apply.
Linux	This rule does not apply.

7.17 *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 very 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.

7.17.1 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 *listen*() 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.

7.17.2 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: *socket_1*; *return_1*; *bind_1 return_1*; $listen_1$; *return_1*; ...

7.17.3 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()`.

7.17.4 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.

7.17.5 Summary

listen_1	**tcp: fast succeed**	Successfully put socket in *LISTEN* state
listen_1b	**tcp: fast succeed**	Successfully update backlog value
listen_1c	**tcp: fast succeed**	Successfully put socket in the *LISTEN* state from any non-{*CLOSED*; *LISTEN*} state on FreeBSD
listen_2	**tcp: fast fail**	Fail with *EINVAL* on WinXP: socket not bound to local port
listen_3	**tcp: fast fail**	Fail with *EINVAL* on Linux or *EISCONN* on WinXP: socket not in *CLOSED* or *LISTEN* state
listen_4	**tcp: fast fail**	Fail with *EADDRINUSE* on Linux: another socket already listening on local port
listen_5	**tcp: fast fail**	Fail with *EINVAL* on BSD: socket shutdown for writing or *bsd_cantconnect* flag set
listen_7	**udp: fast fail**	Fail with *EOPNOTSUPP*: *listen*() called on UDP socket

7.17.6 Rules

listen_1 **tcp: fast succeed** **Successfully put socket in *LISTEN* state**

$$
\begin{array}{l}
(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\
\qquad\qquad\qquad \text{TCP\_Sock}(CLOSED, cb, *)))];\\
\quad listen := listen_0|\!\rangle,\\
\quad SS, MM)
\end{array}
$$

$$\xrightarrow{tid \cdot listen(fd, n)} (h \, \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore, \\ \qquad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis)))]; \\ listen := sid :: listen_0; \\ bound := bound |\!\rangle, \\ SS, MM)$$

$$fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge \\ (bsd\_arch\ h.arch \vee cantrcvmore = \mathbf{F}) \wedge \\ \neg(windows\_arch\ h.arch \wedge IS\_NONE\ ps_1) \wedge \\ (bsd\_arch\ h.arch \implies \mathbf{T}) \wedge \\ p_1 \in \text{autobind}(ps_1, PROTO\_TCP, h, socks \backslash\backslash sid) \wedge \\ (\mathbf{if}\ ps_1 = * \ \mathbf{then}\ bound = sid :: h.bound \ \mathbf{else}\ bound = h.bound) \wedge \\ lis = \langle\!| q0 := [\,]; \\ \quad q := [\,]; \\ \quad qlimit := n |\!\rangle$$

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 $listen_0$. 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 $tid \cdot listen(fd, n)$ transition is made, leaving the thread in state $Ret(OK())$. 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, $listen := sid :: listen_0$, and if autobinding occurred, it is also added to the host's list of bound sockets, *h.bound*, to create a new list *bound*.

Variations

FreeBSD	The *bsd_cantconnect* 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 $listen_2$ (p134).

listen_1b **tcp: fast succeed** **Successfully update backlog value**

$$(h \, \langle\!| ts := ts \oplus (tid \mapsto (Run)_d); \\ socks := socks \oplus \\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore, \\ \qquad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis)))]; \\ listen := listen_0 |\!\rangle, \\ SS, MM)$$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$$\xrightarrow{tid\cdot listen(fd,n)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\ \qquad\qquad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis')))];\\ listen := sid :: listen_0|\!\rangle,\\ SS, MM)\end{array}$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$
$(bsd\_arch\ h.arch \vee cantrcvmore = \mathbf{F}) \wedge$
$lis' = lis\ \langle\!| qlimit := n|\!\rangle$

Description

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 the *LISTEN* state. The host has a list of listening sockets, $listen_0$. The call succeeds.

A $tid\cdot listen(fd, n)$ transition is made, leaving the thread state $Ret(OK())$. 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, $listen := sid :: listen_0$.

listen_1c **tcp: fast succeed** **Successfully put socket in the *LISTEN* state from any non-{*CLOSED*; *LISTEN*} state on FreeBSD**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ socks := socks \oplus\\ \quad [(sid, sock)];\\ listen := listen_0|\!\rangle,\\ SS, MM)\end{array} \xrightarrow{tid\cdot listen(fd,n)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, sock')];\\ listen := sid :: listen_0|\!\rangle,\\ SS', MM)\end{array}$$

$bsd\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$
$sock = \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \text{TCP\_PROTO}(tcp\_sock)) \wedge$
$tcp\_sock.st \notin \{CLOSED; LISTEN\} \wedge$
$sock' = sock\ \langle\!| pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!| st := LISTEN; lis := \uparrow lis|\!\rangle)|\!\rangle \wedge$
$\text{destroy}(i_1, p_1, i_2, p_2) SS\ SS' \wedge$
$lis = \langle\!| q0 := [\,];$
$\quad q := [\,];$
$\quad qlimit := n|\!\rangle$

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 $tid\cdot listen(fd, n)$ 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**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$

$\xrightarrow{tid \cdot listen(fd, n)}$ $(h \ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EINVAL))_{sched\_timer})|\!\rangle, SS, MM)$

$windows\_arch\ h.arch \land$
$fd \in \mathbf{dom}(h.fds) \land$
$fid = h.fds[fd] \land$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \land$
$h.socks[sid] = sock \land$
$\text{proto\_of } sock.pr = PROTO\_TCP \land$
$sock.ps_1 = *$

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**

$(h \ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$
$\xrightarrow{tid \cdot listen(fd, n)}$ $(h \ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } err))_{sched\_timer})|\!\rangle, SS, MM)$

$fd \in \mathbf{dom}(h.fds) \land$
$fid = h.fds[fd] \land$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \land$
$h.socks[sid] = sock \land$
$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \land$
$tcp\_sock.st \notin \{CLOSED; LISTEN\} \land$
$\neg(bsd\_arch\ h.arch) \land$
$(\mathbf{if}\ windows\_arch\ h.arch\ \mathbf{then}$
$\quad err = EISCONN$
$\mathbf{else\ if}\ linux\_arch\ h.arch\ \mathbf{then}$
$\quad err = EINVAL$
$\mathbf{else}$
$\quad \mathbf{F})$

Description

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 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*·*listen*(*fd*, *n*) 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: *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**

$$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot listen(fd,n)} (h \,\langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EADDRINUSE))_{sched\_timer})|\!\rangle, SS, MM)$$

$linux\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge$
$h.socks[sid] = sock \wedge$
$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.st = CLOSED \wedge$
$sock.ps_1 = \uparrow p_1 \wedge$
$(\exists sid'\ sock'\ tcp\_sock'.h.socks[sid'] = sock' \wedge sock'.pr = \text{TCP\_PROTO}(tcp\_sock') \wedge$
$tcp\_sock'.st = LISTEN \wedge sock'.ps_1 = sock.ps_1 \wedge$
$\neg(\exists i_1\ i_1'.i_1 \neq i_1' \wedge sock.is_1 = \uparrow i_1 \wedge sock'.is_1 = \uparrow i_1'))$

Description

On Linux, 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*, 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*·*listen*(*fd*, *n*) transition is made, leaving the thread state *Ret*(FAIL *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**

$$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$$
$$socks := socks \oplus$$
$$[(sid, sock \,\langle\!|cantsndmore := cantsndmore; pr := \text{TCP\_PROTO}(tcp\_sock \,\langle\!|st := st|\!\rangle)|\!\rangle)]|\!\rangle,$$
$$SS, MM)$$

$$\xrightarrow{tid\cdot listen(fd,n)}$$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EINVAL))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|cantsndmore := cantsndmore; pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!|st := st|\!\rangle)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$bsd\_arch\ h.arch \wedge$
$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$st\ \in \{CLOSED; LISTEN\} \wedge$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge$
$(cantsndmore = \mathbf{T} \vee \mathbf{T})$

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 $tid{\cdot}listen(fd, n)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ EINVAL)$.

Variations

Linux	This rule does not apply.
WinXP	This rule does not apply.

listen_7 <u>**udp: fast fail**</u> **Fail with** *EOPNOTSUPP*: *listen*() **called on UDP socket**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot listen(fd,n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EOPNOTSUPP))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge$
$\text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP$

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 $tid{\cdot}listen(fd, n)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ EOPNOTSUPP)$.

Calling *listen*() on a socket for a connectionless protocol (such as UDP) is meaningless and is thus an unsupported (*EOPNOTSUPP*) operation.

7.18 *recv*() (TCP only)

$recv : fd * \mathsf{int} * msgbflag\ \mathsf{list} \to (\mathsf{string} * ((ip * port) * \mathsf{bool})\ \mathsf{option})$

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 $((ip * port) * \text{bool})$ 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).

7.18.1 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.

7.18.2 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*; ...

7.18.3 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 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()`.

7.18.4 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`.

7.18.5 Summary

recv_1	**tcp: fast succeed**	Successfully return data from the socket without blocking
recv_2	**tcp: block**	Block, entering state *Recv2* as not enough data is available
recv_3	**tcp: slow nonurgent succeed**	Blocked call returns from *Recv2* state
recv_4	**tcp: fast fail**	Fail with *EAGAIN*: non-blocking call would block waiting for data
recv_7	**tcp: fast fail**	Fail with *ENOTCONN*: socket not connected
recv_8	**tcp: fast fail**	Fail with pending error
recv_8a	**tcp: slow urgent fail**	Fail with pending error from blocked state
recv_9	**tcp: fast fail**	Fail with *ESHUTDOWN*: socket shut down for reading on WinXP

7.18.6 Rules

recv_1 **tcp: fast succeed** **Successfully return data from the socket without blocking**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(\textbf{implode}\ str, *)))_{sched\_timer});\ socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$((st \in \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING;$
$TIME\_WAIT; CLOSE\_WAIT; LAST\_ACK\} \wedge$
$is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2) \vee$
$(st = CLOSED)) \wedge$
$n = clip\_int\_to\_num\ n_0 \wedge$
$opts = \textbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$

(* 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. *)
$\exists rcvq.$
let $have\_all\_data = (\textbf{length}\ rcvq \geq n)$ **in**
let $have\_enough\_data = (\textbf{length}\ rcvq \geq sf.n(SO\_RCVLOWAT))$ **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**
$(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee cantrcvmore) \wedge$

$str = rcvq \wedge$

$peek = (MSG\_PEEK \in opts) \wedge$
$inline = \mathbf{T} \wedge$
$\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, rcvq) s\ s' \wedge$
$\textbf{length}\ rcvq \leq n$

Description

From thread tid, which is in the Run state, a $recv(fd, n_0, opts_0)$ call is made where out-of-band data is not requested. fd refers to a synchronised TCP socket sid with binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$ 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 $recv()$ 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 $opts_0$, (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 $opts_0$ (the $have\_enough\_data \wedge partial\_data\_ok$ 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 $tid \cdot recv(fd, n_0, opts_0)$ transition is made leaving the thread state $Ret(OK(\textbf{implode}\ str, *))$. If the *MSG_PEEK* flag was set in $opts_0$ 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 $\uparrow(om - \textbf{length}\ str)$.

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.

recv_2 **tcp: block** **Block, entering state** *Recv2* **as not enough data is available**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle,\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$

$$\xrightarrow{tid \cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Recv2(sid, n, opts))_{never\_timer})|\!\rangle,\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$n = clip\_int\_to\_num\ n_0\ \wedge$
$opts = \textbf{list\_to\_set}\ opts_0\ \wedge$
$fd \in \textbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)\ \wedge$
$h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *))\ \wedge$
$st \in \{ESTABLISHED; SYN\_SENT; SYN\_RECEIVED; FIN\_WAIT\_1; FIN\_WAIT\_2\}\ \wedge$

(* We block if not enough (see *recv_1* (p140)) data is available and there is no pending error. *)
$\exists rcvq.$
$\textbf{let}\ blocking = \neg(MSG\_DONTWAIT \in opts \vee ff.b(O\_NONBLOCK))\ \textbf{in}$
$\textbf{let}\ have\_all\_data = (\textbf{length}\ rcvq \geq n)\ \textbf{in}$
$\textbf{let}\ have\_enough\_data = (\textbf{length}\ rcvq \geq sf.n(SO\_RCVLOWAT))\ \textbf{in}$
$\textbf{let}\ partial\_data\_ok = (MSG\_WAITALL \notin opts \vee n > sf.n(SO\_RCVBUF) \vee$
$\quad (\neg(bsd\_arch\ h.arch) \wedge MSG\_PEEK \in opts))\ \textbf{in}$
$blocking\ \wedge$
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee cantrcvmore)\ \wedge$
$es = *\ \wedge$

$peek = \mathbf{T}\ \wedge$
$inline = \mathbf{T}\ \wedge$
$\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, rcvq) s\ s'$

Description

From thread *tid*, which is in the *Run* state, a $recv(fd, n_0, opts_0)$ 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 $sf.n(SO\_RVCLOWAT)$ (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 *Recv2* waiting for data; a $tid \cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state $Recv2(sid, n, opts)$.

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 *MSG_PEEK* flag may be set in $opts_0$.

recv_3 **tcp: slow nonurgent succeed** **Blocked call returns from *Recv2* state**

$$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Recv2(sid, n, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]|\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$
$$\xrightarrow{\tau} (h \,\langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(\textbf{implode}\ str, *)))_{sched\_timer});$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]|\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$((st \in \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING;$
$TIME\_WAIT; CLOSE\_WAIT; LAST\_ACK\} \wedge$
$is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2) \vee$
$st = CLOSED) \wedge$

$\exists rcvq.$
(* We return at last if we now have enough (see *recv_1* (p140)) data available. Pending errors are not checked. *)
let $have\_all\_data = (\textbf{length}\ rcvq \geq n)$ **in**
let $have\_enough\_data = (\textbf{length}\ rcvq \geq sf.n(SO\_RCVLOWAT))$ **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**
$(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee cantrcvmore) \wedge$

$str = (rcvq : char\ \text{list}) \wedge$

$peek = (MSG\_PEEK \in opts) \wedge$
$inline = \mathbf{T} \wedge$
$\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, rcvq) s\ s' \wedge$
$\textbf{length}\ rcvq \leq n$

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 $have\_enough\_data \wedge partial\_data\_ok$ case above); (3) there is urgent data available in the socket's receive queue (the *urgent_data_ahead* cae 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 $Ret(OK(\mathbf{implode}\ str, *))$. 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 $\uparrow(om - \mathbf{length}\ str)$.

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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$
$$\xrightarrow{tid \cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EAGAIN))_{sched\_timer})|\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$n = clip\_int\_to\_num\ n_0 \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *)) \wedge$
$st \in \{ESTABLISHED; SYN\_SENT; SYN\_RECEIVED; FIN\_WAIT\_1; FIN\_WAIT\_2\} \wedge$

$\exists rcvq.$
(* We fail if we would otherwise block (see *recv_2* (p141); these conditions are identical). *)
$\mathbf{let}\ blocking = \neg(MSG\_DONTWAIT \in opts \vee ff.b(O\_NONBLOCK))\ \mathbf{in}$
$\mathbf{let}\ have\_all\_data = (\mathbf{length}\ rcvq \geq n)\ \mathbf{in}$
$\mathbf{let}\ have\_enough\_data = (\mathbf{length}\ rcvq \geq sf.n(SO\_RCVLOWAT))\ \mathbf{in}$
$\mathbf{let}\ partial\_data\_ok = (MSG\_WAITALL \notin opts \vee n > sf.n(SO\_RCVBUF) \vee$
$\quad (\neg(bsd\_arch\ h.arch) \wedge MSG\_PEEK \in opts))\ \mathbf{in}$
$\neg blocking \wedge$
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee cantrcvmore) \wedge$
$(rcvq = [\,] \implies es = *) \wedge$

$peek = \mathbf{T} \wedge$
$inline = \mathbf{T} \wedge$
$\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, rcvq) s\ s'$

Description

From thead *tid*, which is in the *Run* state, a $recv(fd, n_0, opts_0)$ 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 *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 $sf.n(SO\_RVCLOWAT)$ (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; (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 $tid\cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state *Ret*(FAIL *EAGAIN*).

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**.

Variations

FreeBSD	In case (iii) above, the *MSG_PEEK* flag may be set in $opts_0$.

recv_7 **tcp: fast fail** **Fail with *ENOTCONN*: socket not connected**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } ENOTCONN))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = h.socks[sid] \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$(tcp\_sock.st = LISTEN \vee$
$(tcp\_sock.st = CLOSED \wedge sock.cantrcvmore = \mathbf{F})$

)

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 *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 $tid\cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state *Ret*(FAIL *ENOTCONN*).

recv_8 **tcp: fast fail** **Fail with pending error**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$$
$$socks := socks \oplus$$
$$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, \uparrow e, cantsndmore, cantrcvmore, \text{TCP\_PROTO}(tcp\_sock)))]|\!\rangle,$$
$$SS, MM)$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)}$$

$(h \, \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } e))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{TCP\_PROTO}(tcp\_sock)))] |\!\rangle,$
$SS, MM)$

$opts = \textbf{list\_to\_set} \ opts_0 \wedge$
$n = clip\_int\_to\_num \ n_0 \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$((tcp\_sock.st \notin \{CLOSED; LISTEN\} \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2) \vee$
$tcp\_sock.st = CLOSED) \wedge$

(* We fail immediately if there is a pending error and we could not otherwise return data (see *recv_1* (p140)). *)
$\textbf{let} \ rcvq = ([\,] : char \ \mathsf{list}) \ \textbf{in}$
$\textbf{let} \ blocking = \neg(MSG\_DONTWAIT \in opts \vee ff.b(O\_NONBLOCK)) \ \textbf{in}$
$\textbf{let} \ have\_all\_data = (\textbf{length} \ rcvq \geq n) \ \textbf{in}$
$\textbf{let} \ have\_enough\_data = (\textbf{length} \ rcvq \geq sf.n(SO\_RCVLOWAT)) \ \textbf{in}$
$\textbf{let} \ partial\_data\_ok = (MSG\_WAITALL \notin opts \vee n > sf.n(SO\_RCVBUF) \vee$
$(\neg(bsd\_arch \ h.arch) \wedge MSG\_PEEK \in opts)) \ \textbf{in}$
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok)) \wedge$
$(blocking \vee rcvq = [\,]) \wedge$

$es = \textbf{if} \ MSG\_PEEK \in opts \ \textbf{then} \uparrow e \ \textbf{else} \ *$

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 $sf.n(SO\_RVCLOWAT)$ (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 $tid \cdot recv(fd, n_0, opts_0)$ 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 $opts_0$.

recv_8a **tcp: slow urgent fail** **Fail with pending error from blocked state**

$(h \, \langle\!| ts := ts \oplus (tid \mapsto (Recv2(sid, n, opts))_d);$
$socks := socks \oplus$
$[(sid, sock \, \langle\!| es := \uparrow e; pr := \text{TCP\_PROTO}(tcp\_sock) |\!\rangle)] |\!\rangle,$
$SS, MM)$

$\xrightarrow{\tau}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|es := es; pr := \text{TCP\_PROTO}(tcp\_sock)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$rcvq = ([\,] : char\ \mathsf{list}) \wedge$

(* We fail now if there is a pending error and we could not otherwise return data (see *recv_1* (p140)). *)
let $have\_all\_data = (\textbf{length}\ rcvq \geq n)$ **in**
let $have\_enough\_data = (\textbf{length}\ rcvq \geq sock.sf.n(SO\_RCVLOWAT))$ **in**
let $partial\_data\_ok = (MSG\_WAITALL \notin opts \vee n > sock.sf.n(SO\_RCVBUF) \vee$
$(\neg(bsd\_arch\ h.arch) \wedge MSG\_PEEK \in opts))$ **in**
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok)) \wedge$

$(es = \textbf{if}\ MSG\_PEEK \in opts\ \textbf{then} \uparrow e\ \textbf{else}\ *)$

Description

Thread *tid* is blocked in state $Recv2(sid, n, opts)$ where *sid* identifies a socket with pending error $\uparrow e$. 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 $sf.n(SO\_RVCLOWAT)$ (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 $Ret(\text{FAIL}\ e)$. 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 *opts*.

recv_9 **tcp: fast fail** **Fail with *ESHUTDOWN*: socket shut down for reading on WinXP**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|cantrcvmore := \mathbf{T}; pr := \text{TCP\_PROTO}(tcp\_sock)|\!\rangle)]|\!\rangle,$
$SS, MM)$
$\xrightarrow{tid\cdot recv(fd, n, opts)}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ESHUTDOWN))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|cantrcvmore := \mathbf{T}; pr := \text{TCP\_PROTO}(tcp\_sock)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$windows\_arch\ h.arch \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)$

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 $tid \cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state $Ret(\text{FAIL } ESHUTDOWN)$.

Variations

FreeBSD	This rule does not apply.
Linux	This rule does not apply.

7.19 *recv*() (UDP only)

$recv : (fd * \mathsf{int} * msgbflag\ \mathsf{list}) \to (\mathsf{string} * ((ip * port) * \mathsf{bool})\ \mathsf{option})$

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, $(\mathsf{string} * ((ip * port) * \mathsf{bool})\ \mathsf{option})$, 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.

7.19.1 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 *MSG_DONTWAIT* flag being set in the *recv*() flags or the socket's *O_NONBLOCK* flag being set.
EMSGSIZE	The amount of data requested in the *recv*() 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 *MSG_WAITALL* flag is set on a *recv*() call on WinXP.
ESHUTDOWN	On WinXP, a *recv*() 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.

7.19.2 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*;

7.19.3 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 *msgbflag* 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()`.

7.19.4 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 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 Linx message flags are not modelled: `MSG_NOSIGNAL`, `MSG_TRUNC`, and `MSG_ERRQUEUE`.

7.19.5 Summary

recv_11	**udp: fast succeed**	Receive data successfully without blocking
recv_12	**udp: block**	Block, entering *Recv2* state as no datagrams available on socket
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
recv_14	**udp: fast fail**	Fail with *EAGAIN*, *EADDRNOTAVAIL*, or *ENOBUFS*: there are no ephemeral ports left
recv_15	**udp: slow urgent succeed**	Blocked call returns from *Recv2* state with data

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
recv_17	**udp: rc**	Socket shutdown for reading: fail with *ESHUTDOWN* on WinXP or succeed on Linux and FreeBSD
recv_20	**udp: rc**	Successful partial read of datagram on head of socket's receive queue on WinXP
recv_21	**udp: fast succeed**	Read zero bytes of data from an empty receive queue on FreeBSD
recv_22	**udp: fast fail**	Fail with *EINVAL* on WinXP: socket is unbound
recv_23	**udp: rc**	Read ICMP error from receive queue and fail with that error on WinXP
recv_24	**udp: fast fail**	Fail with pending error

7.19.6 Rules

recv_11 **udp: fast succeed** **Receive data successfully without blocking**

$$(h \; \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\; socks := socks \oplus [(sid, sock\; \langle\!|pr := \mathrm{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle,\; SS, MM)$$

$$\xrightarrow{tid \cdot recv(fd, n_0, opts_0)} (h \; \langle\!|ts := ts \oplus (tid \mapsto \big(Ret(OK(\textbf{implode}\;\; data', \uparrow((i_3, ps_3), b)))\big)_{sched\_timer});\; socks := socks \oplus [(sid, sock)]|\!\rangle,\; SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, cantsndmore, cantrcvmore, \mathrm{UDP\_Sock}(rcvq')) \wedge$
$(\neg(linux\_arch\; h.arch) \implies cantrcvmore = \mathbf{F}) \wedge$
$rcvq = (Dgram\_msg(\langle\!| is := i_3; ps := ps_3; data := data|\!\rangle)) :: rcvq'' \wedge$
$n = clip\_int\_to\_num\; n_0 \wedge$
$((\mathbf{length}\;\; data \leq n \wedge data = data') \vee$
$(\mathbf{length}\;\; data > n \wedge data' = TAKE\; n\; data \wedge \mathbf{length}\;\; data' = n \wedge \neg(windows\_arch\; h.arch))) \wedge$
$(windows\_arch\; h.arch \implies b = \mathbf{T}) \wedge$
$opts = \mathbf{list\_to\_set}\;\; opts_0 \wedge$
$rcvq' = (\mathbf{if}\;\; MSG\_PEEK \in opts\;\; \mathbf{then}\;\; rcvq\;\; \mathbf{else}\;\; rcvq'')$

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 $tid \cdot recv(fd, n_0, opts_0)$ transition is made. The thread is left in state $Ret(OK(\textbf{implode}\;\; data', \uparrow(i_3, ps_3)))$, 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).

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

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 *recv_20*.

recv_12 **udp: block** **Block, entering *Recv2* state as no datagrams available on socket**

$$(h_0, SS, MM) \xrightarrow{tid \cdot recv(fd, n_0, opts_0)} (h_0 \, \langle\!|ts := ts \oplus (tid \mapsto (Recv2(sid, n, opts))_{never\_timer}); socks := h_0.socks \oplus [(sid, sock \, \langle\!|ps_1 := \uparrow p_1'|\!\rangle)]; bound := bound|\!\rangle, SS, MM)$$

$$\begin{aligned}
&h_0 = h \, \langle\!| \, ts := ts \oplus (tid \mapsto (Run)_d); \\
&\qquad socks := socks \oplus [(sid, sock)]|\!\rangle \wedge \\
&fd \in \mathbf{dom}(h_0.fds) \wedge \\
&fid = h_0.fds[fd] \wedge \\
&h_0.files[fid] = \text{FILE}(FT_Socket(sid), ff) \wedge \\
&sock = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, *, cantsndmore, \mathbf{F}, \text{UDP_Sock}([\,])) \wedge \\
&p_1' \in \text{autobind}(sock.ps_1, PROTO_UDP, h_0, h_0.socks) \wedge \\
&(\mathbf{if} \; sock.ps_1 = * \; \mathbf{then} \; bound = sid :: h_0.bound \; \mathbf{else} \; bound = h_0.bound) \wedge \\
&\neg((MSG_DONTWAIT \in opts \wedge linux_arch \; h.arch) \vee ff.b(O_NONBLOCK)) \wedge \\
&(bsd_arch \; h.arch \implies \neg(n = 0)) \wedge \\
&n = clip_int_to_num \; n_0 \wedge \\
&opts = \mathbf{list_to_set} \; opts_0
\end{aligned}$$

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 $recv(fd, n_0, opts_0)$ call is made. Because there are no datagrams on the socket's receive queue, the call will block.

A $tid \cdot recv(fd, n_0, opts_0)$ transition will be made, leaving the thread state $Recv2(sid, n, opts)$. 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 $recv()$ 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 $MSG\_DONTWAIT$ 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**

$$(h_0, SS, MM) \xrightarrow{tid \cdot recv(fd, n, opts_0)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EAGAIN))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, s\ \langle\!|es := *; pr := \text{UDP\_Sock}([\,])|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array}$$

$$\begin{array}{l}h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);\\ \qquad socks := socks \oplus\\ \qquad\quad [(sid, s\ \langle\!|\ es := *; pr := \text{UDP\_Sock}([\,])|\!\rangle)]|\!\rangle \wedge\\ fd\ \in \mathbf{dom}(h_0.fds) \wedge\\ fid = h_0.fds[fd] \wedge\\ h_0.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge\\ opts = \mathbf{list\_to\_set}\ \ opts_0 \wedge\\ ((MSG\_DONTWAIT\ \in\ opts \wedge linux\_arch\ h.arch) \vee ff.b(O\_NONBLOCK))\end{array}$$

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 $recv(fd, n, opts_0)$ call is made. The call would block because the socket has an empty receive queue, so the call fails with an $EAGAIN$ error.

A $tid \cdot recv(fd, n, opts_0)$ transition is made, leaving the thread state $Ret(\text{FAIL } EAGAIN)$.

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 $O\_NONBLOCK$ flag is not set but the $MSG\_DONTWAIT$ flag is set in $opts_0$. Also, note that EWOULDBLOCK and $EAGAIN$ are aliased on Linux.

recv_14 **udp: fast fail** **Fail with** $EAGAIN$, $EADDRNOTAVAIL$, **or** $ENOBUFS$**: there are no ephemeral ports left**

$$(h_0, SS, MM) \xrightarrow{tid \cdot recv(fd, n, opts)} (h_0\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } e))_{sched\_timer})|\!\rangle, SS, MM)$$

$$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);$$

$socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, *, *, *, *, *, cantsndmore, cantrcvmore, \text{UDP\_Sock}([\,])))]\rangle\!\rangle \wedge$
$\text{autobind}(*, PROTO\_UDP, h_0, h_0.socks) = \emptyset \wedge$
$e \in \{EAGAIN; EADDRNOTAVAIL; ENOBUFS\} \wedge$
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \textsc{File}(FT\_Socket(sid), ff)$

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 $recv(fd, n, opts)$ call is made. There is no ephemeral port to autobind the socket to, so the call fails with either *EAGAIN*, *EADDRNOTAVAIL* or *ENOBUFS*.

A $tid \cdot recv(fd, n, opts)$ transition is made, leaving the thread state $Ret(\text{FAIL } e)$ where e is one of the above errors.

recv_15 **udp: slow urgent succeed** **Blocked call returns from *Recv2* state with data**

$(h \ \langle\!\langle ts := ts \oplus (tid \mapsto (Recv2(sid, n, opts))_d);$
$socks := socks \oplus$
$[(sid, sock \ \langle\!\langle ps_1 := \uparrow p_1; es := *; pr := \text{UDP\_Sock}(rcvq)\rangle\!\rangle)]\rangle\!\rangle,$
$SS, MM)$
$\xrightarrow{\tau} (h \ \langle\!\langle ts := ts \oplus (tid \mapsto \big(Ret(OK(\mathbf{implode}\ data', \uparrow((i_3, ps_3), b)))\big)_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock \ \langle\!\langle ps_1 := \uparrow p_1; es := *; pr := \text{UDP\_Sock}(rcvq')\rangle\!\rangle)]\rangle\!\rangle,$
$SS, MM)$

$rcvq = (Dgram\_msg(\langle\!\langle is := i_3; ps := ps_3; data := data\rangle\!\rangle)) :: rcvq'' \wedge$
$(rcvq' = \mathbf{if}\ MSG\_PEEK \in opts\ \mathbf{then}\ rcvq\ \mathbf{else}\ rcvq'') \wedge$
$((\mathbf{length}\ data \leq n \wedge data = data') \vee$
$(\mathbf{length}\ data > n \wedge \neg(windows\_arch\ h.arch) \wedge data' = TAKE\ n\ data' \wedge \mathbf{length}\ data' = n)) \wedge$
$(windows\_arch\ h.arch \implies b = \mathbf{T})$

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 $Recv2(sid, n, opts)$.

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 $Ret(OK(\mathbf{implode}\, data', \uparrow((i_3, ps_3), b)))$. 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle, \\ SS, MM)$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EOPNOTSUPP))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle, \\ SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$(MSG\_WAITALL \in opts \wedge windows\_arch\ h.arch)$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, in the *Run* state, a $recv(fd, n_0, opts_0)$ call is made. The *MSG_OOB* or *MSG_WAITALL* flags are set in $opts_0$. The call fails with an *EOPNOTSUPP* error.

A $tid\cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state *Ret*(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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|cantrcvmore := \mathbf{T}; pr := \mathrm{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle, \\ SS, MM)$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(ret))_{sched\_timer}); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|cantrcvmore := \mathbf{T}; pr := \mathrm{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle, \\ SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff) \wedge$
if $windows\_arch\ h.arch$ **then** $ret = \mathrm{FAIL}\ (ESHUTDOWN) \wedge rc = fast\ fail$
else if $bsd\_arch\ h.arch$ **then** $ret = OK(\text{``''}, \uparrow((*, *), b)) \wedge rc = fast\ succeed \wedge$
$sock.es = *$

else if $\mathit{linux\_arch}\ h.\mathit{arch}$ **then**
$\mathit{rcvq} = [\,] \wedge \mathit{ret} = \mathit{OK}(``", \uparrow((*, *), b)) \wedge \mathit{rc} = \mathit{fast\ succeed} \wedge \mathit{sock.es} = *$
else $\mathit{ASSERTION\_FAILURE}$"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 $\mathit{recv}(\mathit{fd}, n_0, \mathit{opts}_0)$ call is made. On FreeBSD and Linux, if the socket has no pending error the call is successfully, returning $(``", \uparrow((*, *), b))$; on WinXP the call fails with an *ESHUTDOWN* error.

A $\mathit{tid}\cdot\mathit{recv}(\mathit{fd}, n_0, \mathit{opts}_0)$ transition is made, leaving the thread state $\mathit{Ret}(\mathit{OK}(``", \uparrow((*, *), b)))$ on FreeBSD and Linux, or $\mathit{Ret}(\text{FAIL } \mathit{ESHUTDOWN})$ 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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d);$$
$$socks := socks \oplus$$
$$[(sid, sock\ \langle\!|pr := \text{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle,$$
$$SS, MM)$$
$$\xrightarrow{lbl}\ (h\ \langle\!|ts := ts \oplus (tid \mapsto \big(Ret(OK(\textbf{implode}\ data', \uparrow((i_3, ps_3), \mathbf{F})))\big)_{sched\_timer});$$
$$socks := socks \oplus$$
$$[(sid, sock)]|\!\rangle,$$
$$SS, MM)$$

$$windows\_arch\ h.arch\ \wedge$$
$$rcvq = (Dgram\_msg(\langle\!|\ is := i_3; ps := ps_3; data := data|\!\rangle)) :: rcvq'' \wedge$$
$$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, cantsndmore, cantrcvmore, \text{UDP\_Sock}(rcvq')) \wedge$$
$$((\exists fd\ ff\ n\ n_0\ opts_0.$$
$$fd \in \textbf{dom}(h.fds) \wedge$$
$$fid = h.fds[fd] \wedge$$
$$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$$
$$(rcvq' = \textbf{if}\ MSG\_PEEK \in (\textbf{list\_to\_set}\ opts_0)\ \textbf{then}\ rcvq\ \textbf{else}\ rcvq'') \wedge$$
$$n = clip\_int\_to\_num\ n_0 \wedge$$
$$n < \textbf{length}\ data \wedge$$
$$data' = TAKE\ n\ data \wedge$$
$$t = Run \wedge$$
$$rc = fast\ succeed \wedge$$
$$lbl = tid\cdot recv(fd, n_0, opts_0)) \vee$$
$$(\exists n\ opts.$$
$$lbl = \tau \wedge$$
$$t = Recv2(sid, n, opts) \wedge$$
$$rc = slow\ urgent\ succeed \wedge$$
$$data' = TAKE\ n\ data \wedge$$
$$n < \textbf{length}\ data \wedge$$
$$rcvq' = \textbf{if}\ MSG\_PEEK \in opts\ \textbf{then}\ rcvq\ \textbf{else}\ rcvq''))$$

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 $recv(fd, n_0, opts_0)$ 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 $tid{\cdot}recv(fd, n_0, opts_0)$ transition is made leaving the thread state $Ret(OK(\textbf{implode}\ data', \uparrow((i_3, ps_3), \mathbf{F})))$ where the **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 $Recv2(sid, n, opts)$ 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 $Ret(OK(\textbf{implode}\ data', \uparrow((i_3, ps_3), \mathbf{F})))$ where the **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 $OK(\textbf{implode}\ data', \uparrow((i_3, p_3), \mathbf{F}))$ where the **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**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ socks := socks \oplus\\ \quad [(sid, sock\ \langle\!|pr := \text{UDP\_Sock}([\,])|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array}$$

$$\xrightarrow{tid\cdot recv(fd, n_0, opts_0)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(\text{“”}, \uparrow((*, *), b))))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, sock\ \langle\!|pr := \text{UDP\_Sock}([\,])|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array}$$

$bsd\_arch\ h.arch\ \wedge$
$fd\ \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)\ \wedge$
$0 = clip\_int\_to\_num\ n_0$

Rule version: \$ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp \$

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 $recv(fd, n_0, opts_0)$ call is made where $n_0 = 0$. The call succeeds, returning the empty string and not specifying an address: $OK(\text{``''}, \uparrow((*, *), b))$.

A $tid \cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state $Ret(OK(\text{``''}, \uparrow((*, *), b)))$.

Variations

Posix	This rule does not apply: see rules *recv_12* and *recv_13*.
Linux	This rule does not apply: see rules *recv_12* and *recv_13*.
WinXP	This rule does not apply: see rules *recv_12* and *recv_13*.

recv_22 **udp: fast fail** **Fail with *EINVAL* on WinXP: socket is unbound**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d);$$
$$socks := socks \oplus$$
$$[(sid, sock\ \langle\!\langle ps_1 := *; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle,$$
$$SS, MM)$$
$$\xrightarrow{tid \cdot recv(fd, n_0, opts_0)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EINVAL))_{sched\_timer});$$
$$socks := socks \oplus$$
$$[(sid, sock\ \langle\!\langle ps_1 := *; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle,$$
$$SS, MM)$$

$$windows\_arch\ h.arch\ \wedge$$
$$fd \in \mathbf{dom}(h.fds)\ \wedge$$
$$fid = h.fds[fd]\ \wedge$$
$$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)$$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd* that is not bound to a local port. A $recv(fd, n_0, opts_0$ call is made from thread *tid* which is in the *Run* state. The call fails with an *EINVAL* error.

A $tid \cdot recv(fd, n_0, opts_0)$ transition is made, leaving the thread state $Ret(\text{FAIL } EINVAL)$.

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**

$(h \,\langle\!|ts := ts \oplus (tid \mapsto (t)_d);$
$socks := socks \oplus$
$[(sid, sock \,\langle\!|pr := \text{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$\xrightarrow{lbl}$

$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } err))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock \,\langle\!|pr := \text{UDP\_Sock}(rcvq')|\!\rangle)]|\!\rangle,$
$SS, MM)$

$windows\_arch\ h.arch \wedge$
$rcvq = (Dgram\_error(\langle\!|\ e := err|\!\rangle)) :: rcvq' \wedge$
$((\exists fd\ n_0\ opts_0\ fid\ ff.t = Run \wedge$
$lbl = tid \cdot recv(fd, n_0, opts_0) \wedge$
$rc = fast\ fail \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)) \vee$
$(\exists n\ opts.t = Recv2(sid, n, opts) \wedge$
$lbl = \tau \wedge$
$rc = slow\ urgent\ fail))$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd*. At the head of the socket's receive queue, *rcvq*, is an ICMP message with error *err*. This rule covers two cases.

In the first, thread *tid* is in the *Run* state and a $recv(fd, n_0, opts_0)$ call is made. The call fails with error *err*, making a $tid \cdot recv(fd, n_0, opts_0)$ transition. This leaves the thread state *Ret*(FAIL *err*), and the socket with the ICMP message removed from its receive queue.

In the second case, thread *tid* is blocked in state $Recv2(sid, n_0, opts_0)$. A τ transition is made, leaving the thread state *Ret*(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**

$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, \uparrow e, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot recv(fd, n_0, opts_0)}$

$(h \,\langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } e))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$

$(\neg linux\_arch\ h.arch \implies \exists p_2.ps_2 = \uparrow p_2) \wedge$
$es = \textbf{if}\ MSG\_PEEK \in opts\ \textbf{then} \uparrow e\ \textbf{else}\ *$

Description

From thread *tid*, which is in the *Run* state, a $recv(fd, n_0, opts_0)$ call is made. *fd* refers to a 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 $tid \cdot recv(fd, n_0, opts_0)$ 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.

7.20 *send*() (TCP only)

$send : fd * (ip * port)$ option $*$ string $*$ $msgbflag$ list $\rightarrow$ 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.

7.20.1 Errors

In addition to errors returned via ICMP (see *deliver_in_icmp_3* (p244)), 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.

7.20.2 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*; ...

7.20.3 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.

7.20.4 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.

7.20.5 Summary

send_1	**tcp: fast succeed**	Successfully send data without blocking
send_2	**tcp: block**	Block waiting for space in socket's send queue
send_3	**tcp: slow nonurgent succeed**	Successfully return from blocked state having sent data
send_3a	**tcp: block**	From blocked state, transfer some data to the send queue and remain blocked
send_4	**tcp: fast fail**	Fail with *EAGAIN*: non-blocking semantics requested and call would block
send_5	**tcp: fast fail**	Fail with pending error
send_5a	**tcp: slow urgent fail**	Fail from blocked state with pending error
send_6	**tcp: fast fail**	Fail with *ENOTCONN* or *EPIPE*: socket not connected
send_7	**tcp: rc**	Fail with *EPIPE* or *ESHUTDOWN*: socket shut down for writing
send_8	**tcp: fast fail**	Fail with *EOPNOTSUPP*: message flag not valid

7.20.6 Rules

send_1 **tcp: fast succeed** **Successfully send data without blocking**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]|\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$

$$\xrightarrow{tid \cdot send(fd, *, \textbf{implode}\ str, opts_0)} \quad \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(\textbf{implode}\ str'')))_{sched\_timer});\\ socks := socks \oplus \\ \quad [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\ \qquad\qquad \text{TCP\_Sock}(st, cb, *)))]|\!\rangle,\\ SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)\end{array}$$

$st \in \{ESTABLISHED; CLOSE\_WAIT\} \wedge$
$opts = \textbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(FT\_Socket(sid), ff) \wedge$

$space \in UNIV \wedge$

$(\{MSG\_PEEK; MSG\_WAITALL\} \cap opts = \emptyset \vee bsd\_arch\ h.arch) \wedge$

$(\textbf{if}\ space \geq \textbf{length}\ str\ \textbf{then}$
$\quad str' = str \wedge str'' = [\,]$
$\textbf{else}$
$\quad (ff.b(O\_NONBLOCK) \vee (MSG\_DONTWAIT \in opts \wedge \neg bsd\_arch\ h.arch)) \wedge$
$\quad (\textbf{if}\ bsd\_arch\ h.arch\ \textbf{then}\ space \geq sf.n(SO\_SNDLOWAT)$
$\quad \textbf{else}\ space > 0) \wedge$
$\quad (str', str'') = SPLIT\ space\ str$
$) \wedge$

$flgs = flgs\ \langle\!|\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F}|\!\rangle \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(flgs, str')s\ s'$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, *, \textbf{implode}\ str, opts_0)$ 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 (p41).

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 *send*() operations on the socket, $sf.n(SO\_SNDLOWAT)$.

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 $tid \cdot send(fd, *, \textbf{implode}\ str, opts_0)$ transition is made, leaving the thread state $Ret(OK(\textbf{implode}\ str''))$. If the data was marked as out-of-band, $MSG\_OOB \in opts$, 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 *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 *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.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

send_2 **tcp: block** **Block waiting for space in socket's send queue**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
$\text{TCP\_Sock}(st, cb, *)))]|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot send(fd, *, \mathbf{implode}\ str, opts_0)}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Send2(sid, *, str, opts))_{never\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
$\text{TCP\_Sock}(st, cb, *)))]|\!\rangle,$
$SS, MM)$

$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\neg((\neg bsd\_arch\ h.arch \wedge MSG\_DONTWAIT \in opts) \vee ff.b(O\_NONBLOCK)) \wedge$

$space \in UNIV \wedge$

$(\{MSG\_PEEK; MSG\_WAITALL\} \cap opts = \emptyset \vee bsd\_arch\ h.arch) \wedge$

$((st \in \{ESTABLISHED; CLOSE\_WAIT\} \wedge$
$space < \mathbf{length}\ str) \vee$
$(linux\_arch\ h.arch \wedge st \in \{SYN\_SENT; SYN\_RECEIVED\}))$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, *, \mathbf{implode}\ str, opts_0)$ 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 (p41)), is less than the length in bytes of the data to be sent, *str*.

The call blocks, leaving the thread state $Send2(sid, *, str, opts)$ via a $tid \cdot send(fd, *, \mathbf{implode}\ str, opts_0)$ transition.

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	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

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Send2(sid, *, str, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]\rangle\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$
$$\xrightarrow{\tau} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK(\mathbf{implode}\ str'')))_{sched\_timer});$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]\rangle\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$st\ \in \{ESTABLISHED; CLOSE\_WAIT\} \wedge$

$space\ \in\ UNIV \wedge$

$space \geq \mathbf{length}\ str \wedge$
$str' = str \wedge str'' = [\,] \wedge$

$flgs = flgs\ \langle\!\langle\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F}\rangle\!\rangle \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(flgs, str')s\ s'$

Description

Thread *tid* is blocked in state $Send2(sid, *, str, opts)$ 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 (p41)), 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 $Ret(OK\text{``''})$. If the data was marked as out-of-band, $MSG\_OOB\ \in\ opts$, 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 *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**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Send2(sid, *, str, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]\rangle\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$$
$$\xrightarrow{\tau} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Send2(sid, *, str'', opts))_{never\_timer});$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *)))]\rangle\!\rangle,$$
$$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$$

$st\ \in \{ESTABLISHED; CLOSE\_WAIT\} \wedge$
$space\ \in\ UNIV \wedge$
$space < \mathbf{length}\ str \wedge space > 0 \wedge$
$(str', str'') = SPLIT\ space\ str \wedge$

$flgs = flgs\ \langle\!\langle\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F} \rangle\!\rangle\ \wedge$
$\mathrm{write}(i_1, p_1, i_2, p_2)(flgs, str')s\ s'$

Description

Thread *tid* is blocked in state $Send2(sid, *, str, opts)$ 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 (p41)), 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 $Send2(sid, *, str'', opts)$ 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 <u>**tcp: fast fail**</u> **Fail with** *EAGAIN***: non-blocking semantics requested and call would block**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d) \rangle\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot send(fd, *, \mathbf{implode}\ str, opts_0)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EAGAIN))_{sched\_timer}) \rangle\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \mathrm{FILE}(FT\_Socket(sid), ff)\ \wedge$
$h.socks[sid] = \mathrm{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
$\quad \mathrm{TCP\_Sock}(st, cb, *))\ \wedge$
$opts = \mathbf{list\_to\_set}\ \ opts_0\ \wedge$

$(\{MSG\_PEEK; MSG\_WAITALL\} \cap\ opts = \emptyset \vee bsd\_arch\ h.arch)\ \wedge$

$((\neg bsd\_arch\ h.arch \wedge MSG\_DONTWAIT\ \in\ opts) \vee ff.b(O\_NONBLOCK))\ \wedge$

$((st\ \in \{ESTABLISHED; CLOSE\_WAIT\}\ \wedge$
$\quad space\ \in\ UNIV\ \wedge$
$\quad \neg(space \geq \mathbf{length}\ str \vee (\mathbf{if}\ bsd\_arch\ h.arch\ \mathbf{then}\ space \geq sf.n(SO\_SNDLOWAT)\ \mathbf{else}\ space > 0))) \vee$
$(st\ \in \{SYN\_SENT; SYN\_RECEIVED\}\ \wedge$
$\quad linux\_arch\ h.arch))$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, *, \mathbf{implode}\ str, opts_0)$ 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 (p41)), 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, $sf.n(SO\_SNDLOWAT)$, 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 $tid\cdot send(fd, *, \mathbf{implode}\ \ str, opts_0)$ transition is made, leaving the thread in state $Ret(\mathrm{FAIL}\ EAGAIN)$.

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	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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle,\ SS, MM)$$

$$\xrightarrow{tid\cdot send(fd, addr, \mathbf{implode}\ str, opts_0)}$$

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle,\ SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{proto\_of}\ sock.pr = PROTO\_TCP$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid\cdot send(fd, addr, \mathbf{implode}\ str, opts)$ transition is made, leaving the thread in state $Ret(\text{FAIL}\ e)$.

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.

send_5a **tcp: slow urgent fail** **Fail from blocked state with pending error**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Send2(sid, *, str, opts))_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle,\ SS, MM) \xrightarrow{\tau} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle,\ SS, MM)$$

$\text{proto\_of}\ sock.pr = PROTO\_TCP$

Description

Thread *tid* is blocked in state $Send2(sid, *, str, opts)$ from an earlier *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 $Ret(\text{FAIL}\ e)$.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

send_6 **tcp: fast fail** **Fail with** *ENOTCONN* **or** *EPIPE*: **socket not connected**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM) \xrightarrow{tid\cdot send(fd, *, \mathbf{implode}\ str, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = (h.socks[sid]) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$sock.es = * \wedge$
$(tcp\_sock.st \in \{CLOSED; LISTEN\} \vee$

$(tcp\_sock.st \in \{SYN\_SENT; SYN\_RECEIVED\} \wedge \neg(linux\_arch\ h.arch)) \vee$
F (* Placeholder for: if `tcp_disconnect` or `tcp_usrclose` has been invoked *)
$) \wedge$
$err = (\mathbf{if}\ linux\_arch\ h.arch\ \mathbf{then}\ EPIPE\ \mathbf{else}\ ENOTCONN)$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, *, \mathbf{implode}\ str, opts_0)$ 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 $tid\cdot send(fd, *, \mathbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state *Ret*(FAIL *err*) where *err* is one of the above errors.

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

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**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d);$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{TCP\_PROTO}(tcp)))]|\!\rangle,$
$SS, MM)$

$$\xrightarrow{lbl} \quad (h \,\langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ err))_{sched\_timer});$$
$$socks := socks \oplus$$
$$[(sid, \mathrm{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \mathrm{TCP\_PROTO}(tcp)))]|\!\rangle,$$
$$SS, MM)$$

$$\left(\begin{pmatrix}\exists fd\ ff\ str\ opts_0\ i_2\ p_2. \\ \quad fd \in \mathbf{dom}(h.fds) \wedge \\ \quad fid = h.fds[fd] \wedge \\ \quad h.files[fid] = \mathrm{File}(FT\_Socket(sid), ff) \wedge \\ \quad t = Run \wedge \\ \quad lbl = tid{\cdot}send(fd, *, \mathbf{implode}\ str, opts_0) \wedge \\ \quad rc = fast\ fail \wedge \\ \quad is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2 \wedge \\ \quad (\mathbf{if}\ tcp.st \neq CLOSED\ \mathbf{then} \\ \qquad \exists i_1\ p_1.is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \\ \qquad \mathbf{else}\ \mathbf{T})\end{pmatrix} \vee \begin{pmatrix}\exists opts\ str. \\ \quad t = Send2(sid, *, str, opts) \wedge \\ \quad lbl = \tau \wedge \\ \quad rc = slow\ urgent\ fail\end{pmatrix}\right) \wedge$$

$$(\mathbf{if}\ windows\_arch\ h.arch\ \mathbf{then}\ err = ESHUTDOWN$$
$$\mathbf{else}\ err = EPIPE)$$

Description

This rule covers two cases: (1) from thread *tid*, which is in the *Run* state, a $send(fd, *, \mathbf{implode}\ str, opts_0)$ call is made; and (2) thread *tid* is blocked in state $Send2(sid, *, str, opts)$. 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 *Ret*(FAIL *EPIPE*), via a $tid{\cdot}send(fd, *, \mathbf{implode}\ str, opts_0)$ transition in (1) or a τ transition in (2).

Model details

The data to be sent is of type string in the *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

WinXP	The call fails with an *ESHUTDOWN* error instead of *EPIPE*.

send_8 **tcp: fast fail** **Fail with *EOPNOTSUPP*: message flag not valid**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid{\cdot}send(fd, *, \mathbf{implode}\ str, opts_0)}$$
$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ EOPNOTSUPP))_{sched\_timer})|\!\rangle, SS, MM)$$

$$fd \in \mathbf{dom}(h.fds) \wedge$$
$$fid = h.fds[fd] \wedge$$
$$h.files[fid] = \mathrm{File}(FT\_Socket(sid), ff) \wedge$$
$$\mathrm{proto\_of}(h.socks[sid]).pr = PROTO\_TCP \wedge$$
$$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$$
$$(MSG\_PEEK \in opts \vee MSG\_WAITALL \in opts) \wedge$$
$$\neg bsd\_arch\ h.arch$$

Description

From thread *tid*, which is in the *Run* state, a $send(fd, *, \textbf{implode}\ str, opts_0)$ 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 $tid \cdot send(fd, *, \textbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state *Ret*(FAIL *EOPNOTSUPP*).

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	This rule does not apply.

7.21 *send*() (UDP only)

$send : (fd * (ip * port)$ option $*$ string $* msgbflag$ list$) \rightarrow$ string

This section describes the behaviour of *send*() for UDP sockets. A call to $send(fd, addr, data, flags)$ 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 is_2 and ps_2 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 i_3, p_3 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.

7.21.1 Errors

In addition to errors returned via ICMP (see *deliver_in_icmp_3* (p244)), 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 *send*() call would block and non-blocking behaviour is requested. This may have been done either via the *MSG_DONTWAIT* flag being set in the *send*() 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 *send*() 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 *disconnect*() 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.

7.21.2 Common cases

send_9; *return_1*;

7.21.3 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 $addr = \uparrow(i_3, p_3)$ 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()`.

7.21.4 Model details

If the call blocks then the thread enters state $Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts)$ 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,
- $is_1 : ip$ option is the socket's local IP address, possibly $*$,
- $ps_1 : port$ option is the socket's local port, possibly $*$,
- $is_2 : ip$ 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.

7.21.5 Summary

send_9	**udp: fast succeed**	Enqueue datagram and return successfully
send_10	**udp: block**	Block waiting to enqueue datagram
send_11	**udp: fast fail**	Fail with *EAGAIN*: call would block and non-blocking behaviour has been requested
send_12	**udp: fast fail**	Fail with *ENOTCONN*: no peer address set in socket and no destination address provided
send_13	**udp: fast fail**	Fail with *EMSGSIZE*: string to be sent is bigger than *UDPpayloadMax*
send_14	**udp: fast fail**	Fail with *EAGAIN*, *EADDRNOTAVAIL* or *ENOBUFS*: there are no ephemeral ports left
send_15	**udp: slow urgent succeed**	Return from blocked state after datagram enqueued
send_16	**udp: slow urgent fail**	Fail: blocked socket has entered an error state
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
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
send_19	**udp: fast fail**	Fail with *EADDRINUSE*: on FreeBSD, local and destination address quad in use by another socket
send_21	**udp: fast fail**	Fail with *EISCONN*: socket has peer address set and destination address is specified in call on FreeBSD
send_22	**udp: fast fail**	Fail with *EPIPE* or *ESHUTDOWN*: socket shut down for writing
send_23	**udp: fast fail**	Fail with pending error

7.21.6 Rules

send_9 **udp: fast succeed** **Enqueue datagram and return successfully**

$$(h_0, SS, MM)$$

$$\xrightarrow{tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)}$$

$$\begin{aligned}
&(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(``")))_{sched_timer});\\
&socks := socks \oplus\\
&\quad [(sid, sock\ \langle\!|es := es; ps_1 := \uparrow p_1'; pr := \mathrm{UDP_PROTO}(udp)|\!\rangle)];\\
&bound := bound;\\
&oq := oq'|\!\rangle,\\
&SS, MM)
\end{aligned}$$

$$\begin{aligned}
&h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);\\
&\qquad socks := socks \oplus\\
&\qquad\quad [(sid, sock\ \langle\!|\ es := es; pr := \mathrm{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle \wedge\\
&fd\ \in \textbf{dom}(h_0.fds) \wedge\\
&fid = h_0.fds[fd] \wedge\\
&h_0.files[fid] = \mathrm{FILE}(FT_Socket(sid), ff) \wedge\\
&sock.cantsndmore = \mathbf{F} \wedge\\
&STRLEN(\textbf{implode}\ str) \leq UDPpayloadMax\ h_0.arch \wedge\\
&((addr \neq *) \vee (sock.is_2 \neq *)) \wedge\\
&p_1'\ \in\ \mathrm{autobind}(sock.ps_1, PROTO_UDP, h_0, h_0.socks) \wedge\\
&(\textbf{if}\ sock.ps_1 = *\ \textbf{then}\ bound = sid :: h_0.bound\ \textbf{else}\ bound = h_0.bound) \wedge\\
&\mathrm{dosend}(h.ifds, h.rttab, (addr, str), (sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{T}) \wedge\\
&(\textbf{if}\ bsd_arch\ h.arch\ \textbf{then}\ (h_0.socks[sid]).sf.n(SO_SNDBUF) \geq STRLEN(\textbf{implode}\ str)\\
&\quad \textbf{else}\ \mathbf{T}) \wedge\\
&(\neg(windows_arch\ h.arch) \implies es = *)
\end{aligned}$$

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 *Run* state, a call $send(fd, addr, \textbf{implode}\ str, opts_0)$ succeeds if:

- the length of *str* is less than *UDPpayloadMax*, 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 dosend (p42).

A $tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state $Ret(OK(``"))$ 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 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

Posix	The *MSG_OOB* flag is not set in $opts_0$.
FreeBSD	On FreeBSD there is an additional condition for a successful *send*(): the amount of data to be sent must be less than or equal to the size of the socket's send buffer.
Linux	The *MSG_OOB* flag is not set in $opts_0$.
WinXP	The *MSG_OOB* flag is not set in $opts_0$ and any pending errors are ignored.

send_10 **udp: block** **Block waiting to enqueue datagram**

$$(h_0, SS, MM)$$

$$\xrightarrow{tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)}$$

$$\begin{array}{l}(h\ \langle\!|ts := \\ ts \oplus (tid \mapsto \\ \qquad (Send2(sid, \uparrow(addr, sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), str, opts))_{never\_timer}); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|es := es; ps_1 := \uparrow p_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]; \\ bound := bound; \\ oq := oq'|\!\rangle, \\ \quad SS, MM)\end{array}$$

$$\begin{array}{l}h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d); \\ \qquad socks := socks \oplus \\ \qquad\quad [(sid, sock\ \langle\!|\ es := es; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle\ \wedge \\ fd\ \in \textbf{dom}(h_0.fds) \wedge \\ fid = h_0.fds[fd] \wedge\end{array}$$

$h_0.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock.cantsndmore = \mathbf{F} \wedge$
$(\neg(windows\_arch\ h.arch) \implies es = *) \wedge$
$opts = \mathbf{list\_to\_set}\ \ opts_0 \wedge$
$\neg((\neg bsd\_arch\ h.arch \wedge MSG\_DONTWAIT \in opts) \vee ff.b(O\_NONBLOCK)) \wedge$
$((linux\_arch\ h.arch \vee windows\_arch\ h.arch) \implies \mathbf{T}) \wedge$

$p_1' \in \text{autobind}(sock.ps_1, PROTO\_UDP, h_0, h_0.socks) \wedge$
$(\mathbf{if}\ sock.ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \wedge$
$\text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F}) \wedge$
$((addr \neq *) \vee (sock.is_2 \neq *))$

Description

Consider a UDP socket *sid* referenced by *fd* that is not shutdown for writing and has no pending errors. A $send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $opts_0$.

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 $sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2$, the destination address argument *addr*, and the data *str*, cannot be placed on the outqueue of the host *oq*.

The 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 *MSG_DONTWAIT* flag may be set in $opts_0$: it is ignored by FreeBSD.
Linux	The *MSG_OOB* flag must not be set in $opts_0$.
WinXP	The *MSG_OOB* 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**

(h_0, SS, MM)

$\xrightarrow{tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EAGAIN))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|es := es; ps_1 := \uparrow p_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)];$
$bound := bound;$
$oq := oq'|\!\rangle,$
$SS, MM)$

$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);$

$socks := socks \oplus$
$[(sid, sock ⦇ es := es; pr := \text{UDP\_PROTO}(udp)⦈)]⦈ \wedge$
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock.cantsndmore = \mathbf{F} \wedge$
$(\neg(windows\_arch\ h.arch) \implies es = *) \wedge$
$p_1' \in \text{autobind}(sock.ps_1, PROTO\_UDP, h_0, h_0.socks) \wedge$
$(\mathbf{if}\ sock.ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \wedge$
$((addr \neq *) \vee (sock.is_2 \neq *)) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$((\neg bsd\_arch\ h.arch \wedge MSG\_DONTWAIT \in opts) \vee ff.b(O\_NONBLOCK)) \wedge$
$\text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F})$

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 $send(fd, addr, \mathbf{implode}\, str, opts_0$ 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 $tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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**

(h_0, SS, MM)

$\xrightarrow{tid \cdot send(fd, *, \mathbf{implode}\ str, opts_0)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1', *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))];$
$bound := bound|\!\rangle,$
$SS, MM)$

$$
\begin{aligned}
&h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);\\
&\quad socks := socks \oplus\\
&\quad\quad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle \wedge\\
&fd\ \in \mathbf{dom}(h.fds) \wedge\\
&fid = h.fds[fd] \wedge\\
&h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge\\
&(\mathbf{if}\ \ bsd\_arch\ h.arch\ \ \mathbf{then}\ \ err = EDESTADDRREQ\\
&\quad \mathbf{else}\ \ err = ENOTCONN) \wedge\\
&(\neg(windows\_arch\ h.arch) \implies es = *) \wedge\\
&(\mathbf{if}\ \ linux\_arch\ h.arch\ \ \mathbf{then}\\
&\quad\quad \exists p_1'.p_1'\ \in\ \text{autobind}(ps_1, PROTO\_UDP, h_0, h_0.socks) \wedge ps_1' = \uparrow p_1' \wedge\\
&\quad\quad (\mathbf{if}\ \ ps_1 = *\ \ \mathbf{then}\ \ bound = sid :: h_0.bound\ \ \mathbf{else}\ \ bound = h_0.bound)\\
&\quad \mathbf{else}\ \ bound = h_0.bound \wedge ps_1' = ps_1)
\end{aligned}
$$

Description

Consider a UDP socket *sid* referenced by *fd* that has no pending errors.

A call $send(fd, addr, \mathbf{implode}\ \ str, opts_0$ 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 $tid\cdot send(fd, *, \mathbf{implode}\ \ str, opts_0)$ 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, $h_0.bound$, 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, $ps_1 = *$, then it will not be autobound.

send_13 **udp: fast fail** **Fail with** ***EMSGSIZE*****: string to be sent is bigger than** *UDPpayloadMax*

(h_0, SS, MM)
$\xrightarrow{tid\cdot send(fd, addr, \mathbf{implode}\ \ str, opts_0)}$ $(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EMSGSIZE))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|ps_1 := ps_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)];$
$bound := bound|\!\rangle,$
$SS, MM)$

$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);$

$socks := socks \oplus$
$[(sid, sock \;\langle\!|\; pr := \mathrm{UDP\_PROTO}(udp)|\!\rangle)]]\rangle \wedge$
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \mathrm{F{\scriptsize ILE}}(FT\_Socket(sid), ff) \wedge$
$(STRLEN(\mathbf{implode}\ str) > UDPpayloadMax\ h_0.arch \vee$
$(bsd\_arch\ h.arch \wedge STRLEN(\mathbf{implode}\ str) > (h_0.socks[sid]).sf.n(SO\_SNDBUF))) \wedge$
$ps_1' \in \{sock.ps_1\} \cup (\mathbf{image}(\uparrow)(\mathrm{autobind}(sock.ps_1, PROTO\_UDP, h_0, h_0.socks))) \wedge$
$(\mathbf{if}\ sock.ps_1 = * \wedge ps_1' \neq *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound)$

Description

Consider a UDP socket *sid* referenced by *fd*. A call $send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 *send*() call may also fail with *EMSGSIZE* if the size of *str* is greater than the value of the socket's *SO_SNDBUF* option.

send_14 **udp: fast fail** **Fail with *EAGAIN*, *EADDRNOTAVAIL* or *ENOBUFS*: there are no ephemeral ports left**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \mathrm{S{\scriptsize OCK}}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \mathrm{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ e))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \mathrm{S{\scriptsize OCK}}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \mathrm{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{F{\scriptsize ILE}}(FT\_Socket(sid), ff) \wedge$
$cantsndmore = \mathbf{F} \wedge$
$(\neg(windows\_arch\ h.arch) \implies es = *) \wedge$
$\mathrm{autobind}(*, PROTO\_UDP, h, h.socks) = \emptyset \wedge$
$e \in \{EAGAIN; EADDRNOTAVAIL; ENOBUFS\}$

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 $send(fd, addr, \textbf{implode}\ str, opts_0)$ 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 $tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)$ 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 *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

WinXP	Any pending error on the socket is ignored.

send_15 **udp: slow urgent succeed** **Return from blocked state after datagram enqueued**

$$
\begin{array}{l}
(h\ \langle\!|ts := ts \oplus (tid \mapsto (Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d);\\
\quad socks := socks \oplus\\
\quad\quad [(sid, sock\ \langle\!|es := es; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\\
\quad SS, MM)\\
\xrightarrow{\tau}\ (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK(``")))_{sched\_timer});\\
\quad socks := socks \oplus\\
\quad\quad [(sid, sock\ \langle\!|es := es; pr := \text{UDP\_PROTO}(udp)|\!\rangle)];\\
\quad oq := oq'|\!\rangle,\\
\quad SS, MM)
\end{array}
$$

$$
\begin{array}{l}
sock.cantsndmore = \mathbf{F}\ \wedge\\
(\neg(windows\_arch\ h.arch) \implies es = *)\ \wedge\\
STRLEN(\textbf{implode}\ str) \leq UDPpayloadMax\ h.arch\ \wedge\\
(\text{dosend}(h.ifds, h.rttab, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T})\ \vee\\
\quad \text{dosend}(h.ifds, h.rttab, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h.oq, oq', \mathbf{T}))\ \wedge\\
(addr \neq * \vee sock.is_2 \neq * \vee is_2 \neq *)
\end{array}
$$

Description

Consider a UDP socket *sid* that is not shutdown for writing and has no pending errors. The thread *tid* is blocked in state $Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

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 *send*() call.
- is_2, ps_2, the socket's peer address when the *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 $Ret(OK(``"))$, and the host with new outqueue oq'.

send_16 **udp: slow urgent fail** **Fail: blocked socket has entered an error state**

$$(h \; \langle\!| ts := ts \oplus (tid \mapsto (Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str))_d); \; socks := socks \oplus [(sid, sock \; \langle\!| es := \uparrow e; pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle, SS, MM)$$

$$\xrightarrow{\tau} (h \; \langle\!| ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ e))_{sched\_timer}); \; socks := socks \oplus [(sid, sock \; \langle\!| es := *; pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle, SS, MM)$$

$\neg(windows\_arch \; h.arch)$

Description

Consider a UDP socket *sid* that has pending error $\uparrow e$. The thread *tid* is blocked in state $Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$. The error, e, will be returned to the caller.

At τ transition is made, leaving the thread state $RET(\mathrm{FAIL}\ e)$.

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 *send*() 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**

$$(h \; \langle\!| ts := ts \oplus (tid \mapsto (Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d); \; socks := socks \oplus [(sid, sock \; \langle\!| sf := sf; es := es; pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle, SS, MM)$$

$$\xrightarrow{\tau} (h \; \langle\!| ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ e))_{sched\_timer}); \; socks := socks \oplus [(sid, sock \; \langle\!| sf := sf; es := es; pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle, SS, MM)$$

$$\begin{aligned}
&(\neg(windows_arch \; h.arch) \implies es = *) \wedge \\
&(\exists oq'. \mathrm{dosend}(h.ifds, h.rttab, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T})) \wedge \\
&((STRLEN(\mathbf{implode}\ str) > UDPpayloadMax\ h.arch \wedge (e = EMSGSIZE)) \vee \\
&\quad (bsd_arch\ h.arch \wedge STRLEN(\mathbf{implode}\ str) > sf.n(SO_SNDBUF) \wedge (e = EMSGSIZE)) \vee \\
&\quad ((sock.is_2 = *) \wedge (addr = *) \wedge (e = ENOTCONN)))
\end{aligned}$$

Description

Consider a UDP socket *sid* with no pending errors. The thread *tid* is blocked in state $Send2(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

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 *Ret*(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**

$$(h_0, SS, MM) \xrightarrow{tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)} (h\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EOPNOTSUPP))_{sched\_timer}); socks := socks \oplus [(sid, sock\ \langle\!| ps_1 := ps_1'; pr := \text{UDP\_PROTO}(udp) |\!\rangle)]; bound := bound |\!\rangle, SS, MM)$$

$$\begin{aligned}
& h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d); \\
& \qquad socks := socks \oplus \\
& \qquad [(sid, sock\ \langle\!|\ ps_1 := ps_1; pr := \text{UDP_PROTO}(udp) |\!\rangle)] |\!\rangle \wedge \\
& fd \in \mathbf{dom}(h.fds) \wedge \\
& fid = h.fds[fd] \wedge \\
& h.files[fid] = \text{File}(FT_Socket(sid), ff) \wedge \\
& opts = \mathbf{list_to_set}\ opts_0 \wedge \\
& (MSG_PEEK \in opts \wedge windows_arch\ h.arch) \wedge \\
& (\mathbf{if}\ linux_arch\ h.arch\ \mathbf{then} \\
& \quad \exists p_1'.p_1' \in \text{autobind}(ps_1, PROTO_UDP, h_0, h_0.socks) \wedge ps_1' = \uparrow p_1' \wedge \\
& \quad (\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \\
& \mathbf{else} \\
& \quad ps_1 = ps_1' \wedge bound = h_0.bound)
\end{aligned}$$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the *Run* state, a $send(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 *send*() call fail with an *EOPNOTSUPP* error.

A $tid \cdot send(fd, addr, \mathbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state *Ret*(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**. 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 *send*().
Linux	Linux ignores the *MSG_PEEK* flag for *send*().

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

send_19 **udp: fast fail** **Fail with *EADDRINUSE*: on FreeBSD, local and destination address quad in use by another socket**

(h_0, SS, MM)

$\xrightarrow{tid \cdot send(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)}$

$(h\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EADDRINUSE))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock)];$
$bound := bound |\!\rangle,$
$SS, MM)$

$bsd\_arch\ h.arch \wedge$
$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, sock)] |\!\rangle \wedge$
$sock.cantsndmore = \mathbf{F} \wedge$
$(\neg(windows\_arch\ h.arch) \implies sock.es = *) \wedge$
$p_1' \in \text{autobind}(sock.ps_1, PROTO\_UDP, h_0, h_0.socks) \wedge$
$(\textbf{if}\ sock.ps_1 = *\ \textbf{then}\ bound = sid :: h_0.bound\ \textbf{else}\ bound = h_0.bound) \wedge$
$i_1' \in auto\_outroute(i_2, sock.is_1, h_0.rttab, h_0.ifds) \wedge$
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = (h_0.socks[sid]) \wedge$
$\text{proto\_of}\ sock.pr = PROTO\_UDP \wedge$
$(\exists sid'.$
$sid' \in \mathbf{dom}(h_0.socks) \wedge$
$\textbf{let}\ s = h_0.socks[sid']\ \textbf{in}$
$s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge$
$s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge$
$\text{proto\_of}\ s.pr = PROTO\_UDP)$

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 $send(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)$ 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 *send*() call will fail with an *EADDRINUSE* error.

A $tid \cdot send(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)$ transition will be made, leaving the thread state $Ret(\text{FAIL}\ EADDRINUSE)$.

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 ad-**

dress is specified in call on FreeBSD

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid\cdot send(fd, \uparrow(i_3, p_3), \textbf{implode}\ str, opts_0)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EISCONN))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, sock\ \langle\!|es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,$
$SS, MM)$

$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$bsd\_arch\ h.arch$

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 $send(fd, \uparrow(i_3, p_3), \textbf{implode}\ str, opts_0)$ 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 $tid\cdot send(fd, \uparrow(i_3, p_3), \textbf{implode}\ str, opts_0)$ transition will be made, leaving the thread state $Ret(\text{FAIL}\ EISCONN)$.

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 *send*() rules.
Linux	This rule does not apply. Linux allows the *send*() call to occur. See the above *send*() rules.
WinXP	This rule does not apply. WinXP allows the *send*() call to occur. See the above *send*() rules.

send_22 **udp: fast fail** **Fail with *EPIPE* or *ESHUTDOWN*: socket shut down for writing**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid\cdot send(fd, addr, \textbf{implode}\ str, opts_0)}$

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ err))_{sched\_timer});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{UDP\_PROTO}(udp)))]|\!\rangle,$
$SS, MM)$

$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\textbf{if}\ windows\_arch\ h.arch\ \textbf{then}\ err = ESHUTDOWN$

else $err = EPIPE$

Description

From thread tid, which is in the Run state, a $send(fd, addr, \textbf{implode}\ str, opts_0)$ 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 $tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state $Ret(\text{FAIL}\ EPIPE)$.

Variations

WinXP	The call fails with an *ESHUTDOWN* error rather than *EPIPE*.

send_23 **udp: fast fail** **Fail with pending error**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle,\ SS, MM)$$

$$\xrightarrow{tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle,\ SS, MM)$$

$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{proto\_of}\ sock.pr = PROTO\_UDP \wedge$
$\neg(windows\_arch\ h.arch)$

Description

From thread tid, which is in the Run state, a $send(fd, addr, \textbf{implode}\ str, opts_0)$ 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 $tid \cdot send(fd, addr, \textbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state $Ret(\text{FAIL}\ e)$.

Variations

WinXP	This rule does not apply: all pending errors are ignored for *send*() calls on WinXP.

7.22 *setfileflags*() (TCP and UDP)

$setfileflags : (fd * filebflag\ \mathsf{list}) \rightarrow \mathsf{unit}$

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.

7.22.1 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.

7.22.2 Common cases

setfileflags_1; *return_1*

7.22.3 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()`.

7.22.4 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.

7.22.5 Summary

setfileflags_1	**all: fast succeed**	Update all the file flags for an open file description

7.22.6 Rules

setfileflags_1 **all: fast succeed** **Update all the file flags for an open file description**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d);\ files := files \oplus [(fid, \textsc{File}(ft, ff\ \langle\!\langle b := ffb \rangle\!\rangle))] \rangle\!\rangle,\ SS, MM)$$

$$\xrightarrow{tid \cdot setfileflags(fd, flags)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ files := files \oplus [(fid, \textsc{File}(ft, ff\ \langle\!\langle b := ffb' \rangle\!\rangle))] \rangle\!\rangle,\ SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$ffb' = \lambda x.x \in flags$

Description

From thread *tid*, which is in the *Run* state, a $setfileflags(fd, flags)$ call is made. *fd* refers to the open file description $(fid, \textsc{File}(ft, ff\ \langle\!\langle b := ffb \rangle\!\rangle))$ 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 $tid \cdot setfileflags(fd, flags)$ 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*.

7.23 *setsockbopt()* (TCP and UDP)

$setsockbopt : (fd * sockbflag * \mathsf{bool}) \to \mathsf{unit}$

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.

7.23.1 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.

7.23.2 Common cases

setsockbopt_1; *return_1*

7.23.3 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 *setsockbopt*().
- `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 *setsockbopt*() where the possible values of `option_name` are limited to: *SO_BSDCOMPAT*, *SO_DONTROUTE*, *SO_KEEPALIVE*, *SO_OOBINLINE*, 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 *setsockbopt*().
- 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()`.

7.23.4 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 *setsockbopt*().
- `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.

7.23.5 Summary

setsockbopt_1	**all: fast succeed**	Successfully set a boolean socket flag
setsockbopt_2	**udp: fast fail**	Fail with *ENOPROTOOPT*: *SO_KEEPALIVE* and *SO_OOBINLINE* options not supported for a UDP socket on WinXP

7.23.6 Rules

setsockbopt_1 **all: fast succeed** **Successfully set a boolean socket flag**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle,\ SS, MM) \xrightarrow{tid\cdot setsockbopt(fd, f, b)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ socks := socks \oplus [(sid, sock')]|\!\rangle,\ SS, MM)$$

$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock' = sock\ \langle\!| sf := sock.sf\ \langle\!| b := sock.sf.b \oplus (f \mapsto b)|\!\rangle|\!\rangle$
$\wedge$

$(windows\_arch\ h.arch \wedge \text{proto\_of}\ sock.pr = PROTO\_UDP$
$\implies f\ \notin \{SO\_KEEPALIVE\})$

Description

Consider a socket *sid*, referenced by *fd*, and with socket flags *sock.sf*. From thread *tid*, which is in the *Run* state, a *setsockbopt*(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*·*setsockbopt*(fd, f, b) is made, leaving the thread state $Ret(OK())$. 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*.

setsockbopt_2 **udp: fast fail** **Fail with** *ENOPROTOOPT*: *SO_KEEPALIVE* **and** *SO_OOBINLINE* **options not supported for a UDP socket on WinXP**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\ SS, MM)$$
$$\xrightarrow{tid\cdot setsockbopt(fd, f, b)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer});\ socks := socks \oplus [(sid, sock\ \langle\!|pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\ SS, MM)$$

$windows\_arch\ h.arch \wedge$
$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$f\ \in \{SO\_KEEPALIVE\}$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the *Run* state, a *setsockbopt*(fd, f, b) call is made, where f is either *SO_KEEPALIVE* or *SO_OOBINLINE*. The call fails with an *ENOPROTOOPT* error.

A *tid*·*setsockbopt*(fd, f, b) transition is made, leaving the thread state *Ret*(FAIL *ENOPROTOOPT*).

Variations

FreeBSD	This rule does not apply.
Linux	This rule does not apply.

7.24 *setsocknopt*() (TCP and UDP)

setsocknopt : (*fd* ∗ *socknflag* ∗ int) → unit

A call *setsocknopt*(*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.

7.24.1 Errors

A call to *setsocknopt*() can fail with the errors below, in which case the corresponding exception is raised:

EINVAL	On FreeBSD, attempting to set a numeric flag to zero.
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.

7.24.2 Common cases

setsocknopt_1; *return_1*

7.24.3 API

setsocknopt() 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 *setsocknopt*().
- `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 *setsocknopt*() 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 *setsocknopt*().
- 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()`.

7.24.4 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 *setsocknopt*().
- `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.

7.24.5 Summary

setsocknopt_1	**all: fast succeed**	Successfully set a numeric socket flag
setsocknopt_2	**all: fast fail**	Fail with *EINVAL*: on FreeBSD numeric socket flags cannot be set to zero
setsocknopt_4	**all: fast fail**	Fail with *ENOPROTOOPT*: *SO_SNDLOWAT* not settable on Linux

7.24.6 Rules

setsocknopt_1 **all: fast succeed** **Successfully set a numeric socket flag**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle,\ SS, MM) \xrightarrow{tid\cdot setsocknopt(fd, f, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ socks := socks \oplus [(sid, sock')]|\!\rangle,\ SS, MM)$$

$$\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \text{FILE}(FT_Socket(sid), ff) \wedge \\
n' = \mathbf{max}(sf_min_n\ h.arch\ f)(\mathbf{min}(sf_max_n\ h.arch\ f)(clip_int_to_num\ n)) \wedge \\
ns = (\mathbf{if}\ \ bsd_arch\ h.arch \wedge f = SO_SNDBUF \wedge n' < sock.sf.n(SO_SNDLOWAT)\ \mathbf{then} \\
\qquad\qquad (sock.sf.n \oplus (f \mapsto n')) \oplus (SO_SNDLOWAT \mapsto n') \\
\qquad \mathbf{else}\ sock.sf.n \oplus (f \mapsto n')) \wedge \\
sock' = sock\ \langle\!|\ sf := sock.sf\ \langle\!|\ n := ns|\!\rangle|\!\rangle
\end{array}$$

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 $setsocknopt(fd, f, n)$ 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 $tid\cdot setsocknopt(fd, f, n)$ 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_SNDBUF* and the new value n is less than the value of the socket's *SO_SNDLOWAT* flag then the *SO_SNDLOWAT* flag is also set to n.

setsocknopt_2 <u>**all: fast fail**</u> **Fail with *EINVAL*: on FreeBSD numeric socket flags cannot be set to zero**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot setsocknopt(fd, f, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EINVAL))_{sched\_timer})|\!\rangle, SS, MM)$$

$clip\_int\_to\_num\ n = 0\ \wedge$
$bsd\_arch\ h.arch$

Description

On FreeBSD, from thread tid, which is in the Run state, a $setsocknopt(fd, f, n)$ 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 $tid \cdot setsocknopt(fd, f, n)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ EINVAL)$.

Variations

Posix	This rule does not apply.
Linux	This rule does not apply.
WinXP	This rule does not apply.

setsocknopt_4 <u>**all: fast fail**</u> **Fail with *ENOPROTOOPT*: *SO_SNDLOWAT* not settable on Linux**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot setsocknopt(fd, f, n)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer})|\!\rangle, SS, MM)$$

$linux\_arch\ h.arch\ \wedge$
$f = SO\_SNDLOWAT$

Description

On Linux, from thread tid, which is in the Run state, a $setsocknopt(fd, f, n)$ call is made. $f = SO\_SNDLOWAT$, which is not settable, so the call fails with an *ENOPROTOOPT* error.

A $tid \cdot setsocknopt(fd, f, n)$ transition is made, leaving the thread state $Ret(\text{FAIL}\ ENOPROTOOPT)$.

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.

7.25 *setsocktopt*() (TCP and UDP)

$setsocktopt : (fd * socktflag * (\mathsf{int} * \mathsf{int})\ \mathsf{option}) \rightarrow \mathsf{unit}$

A call $setsocktopt(fd, f, t)$ 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 $t = \uparrow(s, ns)$ then the timeout is set to s seconds and ns nanoseconds.

7.25.1 Errors

A call to *setsocktopt*() can fail with the errors below, in which case the corresponding exception is raised:

EBADF	The file descriptor *fd* 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 *fd* 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.

7.25.2 Common cases

setsocktopt_1; *return_1*

7.25.3 API

setsocktopt() 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 *setsocktopt*().
- `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 *setsocktopt*() 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 $(\texttt{int} * \texttt{int})$ `option` in the model *setsocktopt*().
- 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()`.

7.25.4 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.
- `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 *setsocknopt*().
- `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.

7.25.5 Summary

setsocktopt_1	**all: fast succeed**	Successfully set a time-option socket flag
setsocktopt_4	**all: fast fail**	Fail with *ENOPROTOOPT*: on WinXP *SO_LINGER* not settable for a UDP socket
setsocktopt_5	**all: fast fail**	Fail with *EDOM*: timeout value too long to fit in socket structure

7.25.6 Rules

setsocktopt_1 <u>**all: fast succeed**</u> **Successfully set a time-option socket flag**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle,\ SS, MM) \xrightarrow{tid \cdot setsocktopt(fd, f, t)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched\_timer});\ socks := socks \oplus [(sid, sock')]|\!\rangle,\ SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$tltimeopt\_wf\ t \wedge$
$t' = time\_of\_tltimeopt\ t \wedge$
$t' \geq 0 \wedge$
$(\mathbf{if}\ f \in \{SO\_RCVTIMEO; SO\_SNDTIMEO\} \wedge t' = 0$
$\ \mathbf{then}\ t'' = \infty$
$\ \mathbf{else}\ t'' = t') \wedge$

$(\textbf{if}\ f = SO\_LINGER \wedge t = \uparrow(s, ns)\ \textbf{then}\ ns = 0\ \textbf{else}\ \mathbf{T}) \wedge$
$(f \in \{SO\_RCVTIMEO; SO\_SNDTIMEO\} \implies t'' = \infty \vee t'' \leq sndrcv\_timeo\_t\_max) \wedge$
$sock' = sock \ (\!|\ sf := sock.sf\ (\!|\ t := sock.sf.t \oplus (f \mapsto t'')|\!)|\!)$

Description

From thread *tid*, which is in the *Run* state, a *setsocktopt*(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·setsocktopt*(fd, f, t) transition is made, leaving the thread state *Ret*(*OK*()). If $t = *$ or $t = \uparrow(0, 0)$ 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.

setsocktopt_4 <u>**all: fast fail**</u> **Fail with *ENOPROTOOPT*: on WinXP *SO_LINGER* not settable for a UDP socket**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$

$\xrightarrow{tid\cdot setsocktopt(fd, f, t)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOPROTOOPT))_{sched\_timer})|\!\rangle, SS, MM)$

$windows\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$\text{proto\_of}(h.socks[sid]).pr = PROTO\_UDP \wedge$
$f = SO\_LINGER$

Description

On WinXP, from thread *tid*, which is in the *Run* state, a *setsocktopt*(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·setsocktopt*(fd, f, t) transition is made, leaving the thread state *Ret*(FAIL *ENOPROTOOPT*).

Variations

FreeBSD	This rule does not apply.
Linux	This rule does not apply.

setsocktopt_5 <u>**all: fast fail**</u> **Fail with *EDOM*: timeout value too long to fit in socket structure**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$

$\xrightarrow{tid\cdot setsocktopt(fd, f, t)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EDOM))_{sched\_timer})|\!\rangle, SS, MM)$

$f \in \{SO\_RCVTIMEO; SO\_SNDTIMEO\} \wedge$
$tltimeopt\_wf\ t \wedge$
$t' = time\_of\_tltimeopt\ t \wedge$
$(\textbf{if}\ t' = 0$
$\ \textbf{then}\ t'' = \infty$

else $t'' = t'$) $\wedge$
$\neg(t'' = \infty \vee t'' \leq sndrcv\_timeo\_t\_max)$

Description

From thread *tid*, which is currently in the *Run* state, a *setsocktopt*(*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 *sndrcv_timeo_t_max*.

A *tid*·*setsocktopt*(*fd*, *f*, *t*) call is made, leaving the thread state *Ret*(FAIL *EDOM*).

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.

7.26 *shutdown*() (TCP and UDP)

shutdown : (*fd* ∗ bool ∗ bool) → 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.

7.26.1 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.

7.26.2 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*

7.26.3 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 $\mathit{shutdown}(\mathit{fd}, \mathbf{T}, \mathbf{F})$ 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 $\mathit{shutdown}(\mathit{fd}, \mathbf{F}, \mathbf{T})$ 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 $\mathit{shutdown}(\mathit{fd}, \mathbf{T}, \mathbf{T})$ 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.

7.26.4 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 $(\mathbf{T}, \mathbf{T})$, $(\mathbf{T}, \mathbf{F})$, $(\mathbf{F}, \mathbf{T})$, and $(\mathbf{F}, \mathbf{F})$. The first three correspond to the allowed values of `how`: `SHUT_RD`, `SHUT_WR`, and `SHUT_RDWR`. The last possible value, $(\mathbf{F}, \mathbf{F})$, is not allowed by Posix, but the model allows a $\mathit{shutdown}(\mathit{fd}, \mathbf{F}, \mathbf{F})$ 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.

7.26.5 Summary

shutdown_1	**tcp: fast succeed**	Shut down read or write half of TCP connection
shutdown_2	**udp: fast succeed**	Shutdown UDP socket for reading, writing, or both
shutdown_3	**tcp: fast fail**	Fail with *ENOTCONN*: cannot shutdown a socket that is not connected on Linux and WinXP
shutdown_4	**udp: fast fail**	Fail with *ENOTCONN*: socket's peer address not set on Linux

7.26.6 Rules

shutdown_1 **tcp: fast succeed** **Shut down read or write half of TCP connection**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathit{Run})_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle,\ SS, MM) \xrightarrow{tid \cdot \mathit{shutdown}(fd, r, w)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathit{Ret}(\mathit{OK}())))_{sched\_timer});\ socks := socks \oplus [(sid, sock')]|\!\rangle,\ SS, MM)$$

$sock = \mathrm{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, pr) \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{File}(FT\_Socket(sid), ff) \wedge$
$pr = \mathrm{TCP\_PROTO}\ tcp\_sock \wedge$
if $bsd\_arch\ h.arch \wedge tcp\_sock.st \in \{CLOSED; LISTEN\} \wedge w$ **then**

Rule version: \$ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp \$

$$\begin{array}{l}
\textbf{let } sock'' = (\text{tcp_close } h.arch\ sock) \textbf{ in} \\
sock' = sock'' \langle\!\langle\ cantsndmore := (w \vee cantsndmore); \\
\quad cantrcvmore := (r \vee cantrcvmore); \\
\quad pr := \text{TCP_PROTO}(\text{tcp_sock_of } sock'' \\
\qquad \langle\!\langle\ lis := *\rangle\!\rangle) \\
\rangle\!\rangle \\
\textbf{else} \\
(\neg bsd_arch\ h.arch \implies \exists i_1\ p_1\ i_2\ p_2.tcp_sock.st = ESTABLISHED \wedge is_1 = \uparrow i_1 \wedge \\
\quad ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2 \wedge tcp_sock.lis = *) \wedge \\
pr' = pr \wedge \\
sock' = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, w \vee cantsndmore, r \vee cantrcvmore, pr')
\end{array}$$

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 suceeds: a $tid{\cdot}shutdown(fd, r, w)$ 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 $cantrcvmore = \mathbf{T}$ and emptying the socket's receive queue; if $w = \mathbf{T}$ then the write-half of the connection is shut down, setting $cantsndmore = \mathbf{T}$; 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 (p52). 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**

$$\begin{array}{l}
(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d); \\
socks := socks \oplus \\
\quad [(sid, sock\ \langle\!\langle cantrcvmore := cantrcvmore; \\
cantsndmore := cantsndmore; \\
pr := \text{UDP_PROTO}(udp_pr)\rangle\!\rangle)]\rangle\!\rangle, \\
SS, MM) \\
\xrightarrow{tid\cdot shutdown(fd, r, w)} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(OK()))_{sched_timer}); \\
socks := socks \oplus \\
\quad [(sid, sock\ \langle\!\langle cantrcvmore := (r \vee cantrcvmore); \\
cantsndmore := (w \vee cantsndmore); \\
pr := \text{UDP_PROTO}(udp_pr)\rangle\!\rangle)]\rangle\!\rangle, \\
SS, MM)
\end{array}$$

$$\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \text{FILE}(FT_Socket(sid), ff) \wedge \\
(linux_arch\ h.arch \implies sock.is_2 \neq *)
\end{array}$$

Description

Consider a UDP socket *sid*, referenced by *fd*. From thread *tid*, which is in the *Run* state, a *shutdown*(*fd*, *r*, *w*) call is made and succeeds.

A *tid*·*shutdown*(*fd*, *r*, *w*) transition is made, leaving the thread state *Ret*(*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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot shutdown(fd, r, w)} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } ENOTCONN))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(FT\_Socket(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
$tcp\_sock.st \neq ESTABLISHED \wedge$
$\neg(bsd\_arch\ h.arch)$

Description

From thread *tid*, which is in the *Run* state, a *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*·*shutdown*(*fd*, *r*, *w*) transition is made, leaving the thread state *Ret*(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**

$$\begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);\\ \quad socks := socks \oplus\\ \quad\quad [(sid, sock\ \langle\!|is_2 := *; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\\ \quad SS, MM)\end{array}$$
$$\xrightarrow{tid \cdot shutdown(fd, r, w)} \begin{array}{l}(h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } ENOTCONN))_{sched\_timer});\\ socks := socks \oplus\\ \quad [(sid, sock\ \langle\!|is_2 := *;\\ cantsndmore := (w \vee sock.cantsndmore);\\ cantrcvmore := (r \vee sock.cantrcvmore);\\ pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle,\\ SS, MM)\end{array}$$

$linux\_arch\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$

$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff)$

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 $shutdown(fd, r, w)$ call is made, and fails with an *ENOTCONN* error.

A $tid \cdot shutdown(fd, r, w)$ transition is made, leaving the thread state *Ret*(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 *shutdown_2*.
WinXP	This rule does not apply: see rule *shutdown_2*.

7.27 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.

7.27.1 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.

7.27.2 Common cases

TCP: *socket_1*; *return_1*; *connect_1*; ... UDP: *socket_1*; *return_1*; *bind_1*; *return_1*; *send_9*; ...

7.27.3 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_STREAMMM*) in the model interface, would be a `socket(AF_INET,SOCK_STREAM,0)` call in Posix; a call to socket(*SOCK_DGRAMMM*) 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.

7.27.4 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.

7.27.5 Summary

socket_1	**all: fast succeed**	Successfully return a new file descriptor for a fresh socket
socket_2	**all: fast fail**	Fail with *EMFILE*: out of file descriptors for this process

7.27.6 Rules

socket_1 **all: fast succeed** **Successfully return a new file descriptor for a fresh socket**

$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d);$
$fds := fds;$
$files := files;$
$socks := socks|\!\rangle,$
$SS, MM)$

$\xrightarrow{tid \cdot (\mathsf{socket}(socktype))}$ $(h\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(OK\ fd))_{sched\_timer});$
$fds := fds';$
$files := files \oplus [(fid, \text{FILE}(FT\_Socket(sid), ff\_default))];$
$socks := socks \oplus [(sid, sock)] |\!\rangle,$
$SS, MM)$

$\mathbf{card}(\mathbf{dom}(fds)) < OPEN\_MAX \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin (\mathbf{dom}(socks)) \wedge$
$nextfd\ h.arch\ fds\ fd \wedge$
$fds' = fds \oplus (fd, fid) \wedge$
$(\mathbf{case}\ socktype\ \mathbf{of}$
$\quad SOCK\_DGRAM \rightarrow (sock =$
$\quad\quad \text{SOCK}(\uparrow fid, sf\_default\ h.arch\ socktype, *, *, *, *, *, \mathbf{F}, \mathbf{F}, \text{UDP\_Sock}([\,]))) \parallel$
$\quad SOCK\_STREAM \rightarrow (sock =$
$\quad\quad \text{SOCK}(\uparrow fid, sf\_default\ h.arch\ socktype, *, *, *, *, *, \mathbf{F}, \mathbf{F},$
$\quad\quad\quad \text{TCP\_Sock}(CLOSED, \text{initial\_cb}, *))))$

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 (p43) as its control block, and all other fields uninitialised; if *socktype* = *SOCK_DGRAM* then a new, unitialised 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*) 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 *fid* to the file description of the socket; and its finite map of file descriptors is updated, adding a mapping from *fd* to *fid*.

A *tid*·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**

$(h\ \langle\!| ts := ts \oplus (tid \mapsto (Run)_d) |\!\rangle, SS, MM)$
$\xrightarrow{tid \cdot (\mathsf{socket}(s))}$ $(h\ \langle\!| ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ EMFILE))_{sched\_timer}) |\!\rangle, SS, MM)$

$\mathbf{card}(\mathbf{dom}(h.fds)) \geq OPEN\_MAX$

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*·socket(*s*) transition is made, leaving the thread state *Ret*(FAIL *EMFILE*).

7.28 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.

7.28.1 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.

7.28.2 Summary

return_1	**all: misc nonurgent**	Return result of system call to caller
badf_1	**all: fast fail**	Fail with *EBADF*: not a valid file descriptor
notsock_1	**all: fast fail**	Fail with *ENOTSOCK*: file descriptor not a valid socket
intr_1	**all: slow nonurgent fail**	Fail with *EINTR*: blocked system call interrupted by signal
resourcefail_1	**all: fast badfail**	Fail with *ENFILE*, *ENOBUFS* or *ENOMEM*: out of resources
resourcefail_2	**all: slow nonurgent badfail**	Fail with *ENFILE*, *ENOBUFS* or *ENOMEM*: from a blocked state with out of resources

7.28.3 Rules

return_1 **all: misc nonurgent** **Return result of system call to caller**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret\ v)_d)\rangle\!\rangle, SS, MM) \xrightarrow{\overline{tid \cdot v}} (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_{never\_timer})\rangle\!\rangle, SS, MM)$$

T

Description

A system call from thread *tid* has completed, leaving the thread state $(Ret\ v)_d$. 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 $(Run)_{never\_timer}$.

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

badf_1 **all: fast fail** **Fail with *EBADF*: not a valid file descriptor**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot opn} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ e))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd\_op\ fd\ opn\ \wedge$
$fd\ \notin \mathbf{dom}(h.fds)\ \wedge$
$(\mathbf{if}\ \ windows\_arch\ h.arch\ \ \mathbf{then}\ \ e = ENOTSOCK\ \ \mathbf{else}\ \ e = EBADF)$

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 *Ret*(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 *sockatmark*(). See the definition of *fd_op*.

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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (Run)_d)|\!\rangle, SS, MM)$$
$$\xrightarrow{tid\cdot opn} (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\text{FAIL}\ ENOTSOCK))_{sched\_timer})|\!\rangle, SS, MM)$$

$fd\_sockop\ fd\ opn\ \wedge$
$fd\ \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(ft, ff)\ \wedge$
$\neg(\exists sid.ft = FT\_Socket(sid))$

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 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 *Ret*(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 *sockatmark*(). See the definition of *fd_sockop*.

intr_1 **all: slow nonurgent fail** **Fail with *EINTR*: blocked system call interrupted by signal**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (st)_d)\rangle\!\rangle, SS, MM)$$
$$\xrightarrow{\tau}\ (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } EINTR))_{sched\_timer})\rangle\!\rangle, SS, MM)$$

$sock = (h.socks[sid]) \wedge$
$(st = Close2(sid) \vee$
$st = Connect2(sid) \vee$
$st = Recv2(sid, n, opts) \vee$
$st = Send2(sid, addr, str, opts) \vee$
$st = PSelect2(readfds, writefds, exceptfds) \vee$
$st = Accept2(sid))$

Description

If on socket *sid* as user call blocked leaving a thread in one of the states: $Close2(sid)$, $Connect2(sid)$, $Recv2(sid)$, $Send2(sid)$, $PSelect2(sid)$ or $Accept2(sid)$ 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**

$$(h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Run)_d)\rangle\!\rangle, SS, MM)$$
$$\xrightarrow{tid \cdot call}\ (h\ \langle\!\langle ts := ts \oplus (tid \mapsto (Ret(\text{FAIL } e))_{sched\_timer})\rangle\!\rangle, SS, MM)$$

$\neg INFINITE\_RESOURCES \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(FT\_Socket(sid), ff) \wedge$
$sock = (h.socks[sid]) \wedge$
$((call = \mathsf{socket}(socktype) \wedge e \in \{ENFILE; ENOBUFS; ENOMEM\}) \vee$
$(call = bind(fd, is_1, ps_1) \wedge e = ENOBUFS) \vee$
$(call = connect(fd, i_2, \uparrow p_2) \wedge e = ENOBUFS) \vee$
$(call = listen(fd, n) \wedge e = ENOBUFS) \vee$
$(call = recv(fd, n, opts) \wedge e \in \{ENOMEM; ENOBUFS\}) \vee$
$(call = getsockname(fd) \wedge e = ENOBUFS) \vee$
$(call = getpeername(fd) \wedge e = ENOBUFS) \vee$
$(call = shutdown(fd, r, w) \wedge e = ENOBUFS) \vee$
$(call = accept(fd) \wedge e \in \{ENFILE; ENOBUFS; ENOMEM\}$
$\wedge \text{proto\_of } sock.pr = PROTO\_TCP))$

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**

$$(h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d)|\!\rangle, SS, MM) \quad \xrightarrow{\tau} \quad (h\ \langle\!|ts := ts \oplus (tid \mapsto (Ret(\mathrm{FAIL}\ e))_{sched\_timer})|\!\rangle, SS, MM)$$

$\neg INFINITE\_RESOURCES \wedge$
$sock = (h.socks[sid]) \wedge$
$((t = Accept2(sid) \wedge e \in \{ENFILE; ENOBUFS; ENOMEM\}) \vee$
$(t = Connect2(sid) \wedge e = ENOBUFS) \vee$
$(t = Recv2(sid, n, opts) \wedge e \in \{ENOBUFS; ENOMEM\}))$

Description

If thread *tid* of host *h* is in state *Accept2*(*sid*), *Connect2*(*sid*) or *Recv2*(*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 8

Host LTS: TCP Input Processing

8.1 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, *ACK*s, and *FIN*s. 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 *RST*s and a variety of pathological situations.

8.1.1 Summary

deliver_in_1	**tcp: network nonurgent**	Passive open: receive SYN, send SYN,ACK
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)
deliver_in_3	**tcp: network nonurgent**	Receive data, FINs, and ACKs in a connected state
di3_topstuff		*deliver_in_3* initial checks
di3_newackstuff		*deliver_in_3* new ack processing, used in di3_ackstuff
di3_ackstuff		*deliver_in_3* ACK processing
di3_datastuff		*deliver_in_3* data processing
di3_ststuff		*deliver_in_3* TCP state change processing
di3_socks_update		*deliver_in_3* socket update processing
deliver_in_3b	**tcp: network nonurgent**	Receive data after process has gone away
deliver_in_4	**tcp: network nonurgent**	Receive and drop (silently) a non-sane or martian segment
deliver_in_5	**tcp: network nonurgent**	Receive and drop (maybe with RST) a sane segment that does not match any socket
deliver_in_7	**tcp: network nonurgent**	Receive RST and zap non-{*CLOSED*; *LISTEN*; *SYN_SENT*; *SYN_RECEIVED*; *TIME_WAIT*} socket
deliver_in_7a	**tcp: network nonurgent**	Receive RST and zap *SYN_RECEIVED* socket
deliver_in_7b	**tcp: network nonurgent**	Receive RST and ignore for *LISTEN* socket

deliver_in_7c	**tcp: network nonurgent**	Receive RST and ignore for *SYN_SENT*(unacceptable ack) or *TIME_WAIT* socket
deliver_in_7d	**tcp: network nonurgent**	Receive RST and zap *SYN_SENT*(acceptable ack) socket
deliver_in_8	**tcp: network nonurgent**	Receive SYN in non-{*CLOSED*; *LISTEN*; *SYN_SENT*; *TIME_WAIT*} state
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

8.1.2 Rules

deliver_in_1 **tcp: network nonurgent** **Passive open: receive SYN, send SYN,ACK**

$(h \, \langle\!|socks := socks \oplus [(sid, sock)];$
$iq := iq;$
$oq := oq|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$

$\xrightarrow{\tau}$

$(h \, \langle\!|socks := socks' \oplus$
(* Listening socket *)
$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(LISTEN, cb, \uparrow lis')));$
(* New socket formed by the incoming SYN *)
$(sid', \text{SOCK}(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(SYN\_RECEIVED, cb'', *)))];$
$iq := iq';$
$oq := oq'|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$

(* **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. *)

$sid \notin (\mathbf{dom}(socks)) \wedge$
$sid' \notin (\mathbf{dom}(socks)) \wedge$
$sid \neq sid' \wedge$

(* 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. *)
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\,\mathbf{F}(iflgs, idata)s\ s' \wedge$
$iflgs = iflgs \, \langle\!| \, SYN := \mathbf{T}; SYNACK := \mathbf{F}; RST := \mathbf{F}|\!\rangle \wedge$
$idata \in UNIV \wedge$

(* 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 *)
$i_1 \in local\_ips \ h.ifds \wedge$
$\neg(is\_broadormulticast \ h.ifds \ i_1) \wedge$
$\neg(is\_broadormulticast \ h.ifds \ i_2) \wedge$

(* Find the socket *sock* that has the best match for the address quad in segment *seg*, see tcp_socket_best_match (p38). Socket *sock* must have a form matching the patten $\text{SOCK}(\ldots)$. *)
$\text{tcp\_socket\_best\_match} \ socks(sid, sock)seg \ h.arch \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$

$\text{TCP\_Sock}(\mathit{LISTEN}, cb, \uparrow lis)) \wedge$

(* 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 $*$. *)
$((is_2 = * \wedge ps_2 = *) \vee$
$(\mathit{bsd\_arch}\ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

(* 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* (p211) and not here. *)
$(\textbf{case}\ is_1\ \textbf{of}\ \uparrow i_1' \rightarrow i_1' = i_1 \parallel * \rightarrow \mathbf{T}) \wedge$
$\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$

(* 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. *)
$\neg(\exists(sid, sock) :: socks.$
$\quad \exists tcp\_sock.$
$\quad sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\quad tcp\_sock.st = \mathit{TIME\_WAIT} \wedge$
$\quad sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2 \wedge$
$\quad \mathbf{F}) \wedge$

(* 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. *)
$socks' = \$o\_f(\lambda sock.$
$\quad \textbf{if}\ \exists tcp\_sock.sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\qquad tcp\_sock.st = \mathit{TIME\_WAIT} \wedge$
$\qquad sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge$
$\qquad sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2$
$\quad \textbf{then}$
$\qquad \text{tcp\_close}\ h.arch\ sock$
$\quad \textbf{else}$
$\qquad sock$
$)socks \wedge$

(* Accept the new connection attempt to the incomplete connection queue if the queue of completed (established) connections is not already full *)

$\mathit{accept\_incoming\_q0}\ lis\ \mathbf{T} \wedge$

(* 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. *)
$(\textbf{choose}\ drop :: \mathit{drop\_from\_q0}\ lis.$
$\quad \textbf{if}\ drop\ \textbf{then}$
$\qquad \exists q0L\ sid''\ q0R.$
$\qquad lis.q0 = q0L \mathbin{@} (sid'' :: q0R) \wedge$
$\qquad q_0' = q0L \mathbin{@} q0R$
$\quad \textbf{else}$
$\qquad q_0' = lis.q0$
$) \wedge$

(* Put the new incomplete connection on the (possibly pruned) incomplete connections queue. *)
$lis' = lis \ (\!| \ q0 := sid' :: q_0' |\!) \wedge$

(* Create a SYN,ACK segment in reply: *)
$rcvbufsize' \ \in \ UNIV \wedge sndbufsize' \ \in \ UNIV \wedge$

(* Store the new receive and send buffer sizes *)
$sf' = sf \ (\!| \ n := funupd\_list \ sf.n[(SO\_RCVBUF, rcvbufsize'); (SO\_SNDBUF, sndbufsize')] |\!) \wedge$

(* Update the new connection's control block in light of above. *)
$cb' = cb \ (\!|$

$\quad tt\_keep := \uparrow((())_{slow\_timer \ TCPTV\_KEEP\_IDLE})$
$|\!) \wedge$

(* Construct the SYN,ACK segment using the values stored in the updated control block for the new connection. *)
$oflgs = oflgs \ (\!| \ SYN := \mathbf{F}; SYNACK := \mathbf{T}; FIN := \mathbf{F}; RST := \mathbf{F} |\!) \wedge$
$odata \ \in \ UNIV \wedge$
$\mathrm{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s' \ s''$

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 *listen*() 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 *SYN* 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_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)**

$(h\ \langle\!|socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es,$
$cantsndmore, cantrcvmore, \text{TCP\_PROTO}\ tcp\_sock))];$
$iq := iq;$
$oq := oq|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$
$\xrightarrow{\tau}$ $(h\ \langle\!|socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es,$
$cantsndmore, cantrcvmore',$
$\text{TCP\_Sock}(st', cb'', *)))];$
$iq := iq';$
$oq := oq'|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$

$tcp\_sock = \text{TCP\_Sock0}(SYN\_SENT, cb, *) \wedge$

$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$

$(iflgs.RST = \mathbf{F} \wedge (iflgs.SYN = \mathbf{T} \vee iflgs.SYNACK = \mathbf{T})) \wedge$

$rcvbufsize' \in UNIV \wedge sndbufsize' \in UNIV \wedge$
$sf' = sf\ \langle\!|\ n := funupd\_list\ sf.n[(SO\_RCVBUF, rcvbufsize');$
$(SO\_SNDBUF, sndbufsize')]|\!\rangle \wedge$

(* softerror may be cleared during an active open *)
(**if** $iflgs.SYNACK$ **then** $t\_softerror' = * \vee t\_softerror' = cb.t\_softerror$
else $t\_softerror' = cb.t\_softerror) \wedge$

(* 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). *)
$data' = idata \wedge$
$FIN' = iflgs.FIN \wedge$

$cb' = cb\ \langle\!|$
$tt\_keep := \uparrow((())_{slow\_timer\ TCPTV\_KEEP\_IDLE});$
$t\_softerror := t\_softerror'$
$|\!\rangle \wedge$

$(oflgs, odata) \in$ (**if** $iflgs.SYNACK$ **then** null_flgs_data
else (**if** $bsd\_arch\ h.arch$ **then** null_flgs_data
else make_syn_ack_flgs_data)) $\wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s'\ s'' \wedge$

$stream\_enqueue\_or\_fail\ \mathbf{T}\ h.arch\ h.rttab\ h.ifds(\uparrow i_1, \uparrow i_2)cb'\ cb'' \wedge$

(* N.B. the flags are already written to the stream during the sync *)

(* Note that we change state even if enqueuing or routing returned an error, trusting to retransmit to solve our problem. *)
(**if** $iflgs.SYNACK$ **then**
(* completion of active open *)
(**if** $\neg FIN'$ **then**
$(cantrcvmore' = cantrcvmore \wedge$
$st' \in$
(**if** $cantsndmore = \mathbf{F}$ **then**
$\{ESTABLISHED\}$

else $\{FIN\_WAIT\_2; FIN\_WAIT\_1\}$)) (* 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

$(cantrcvmore' = \mathbf{T} \wedge$

$st' =$

$(\mathbf{if}\ cantsndmore = \mathbf{F}\ \mathbf{then}$

$CLOSE\_WAIT$

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 *)

$(\mathbf{if}\ \neg FIN'\ \mathbf{then}$

$(st' = SYN\_RECEIVED \wedge$

$cantrcvmore' = cantrcvmore)$

else

$(st' = CLOSE\_WAIT \wedge$ (* yes, really! (in BSD) even though we've not yet had our initial SYN acknowledged! See `tcp_input.c:2065 +/-2000` *)

$cantrcvmore' = \mathbf{T}))$

)

---

deliver_in_3 **tcp: network nonurgent** **Receive data, FINs, and ACKs in a connected state**

$(h\ \langle\!|socks := socks \oplus [(sid, sock)];$
$iq := iq;$
$oq := oq;$
$bndlm := bndlm|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$

$\xrightarrow{\tau}$

$(h\ \langle\!|socks := socks';$
$iq := iq';$
$oq := oq';$
$bndlm := bndlm'|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$

$sid \notin (\mathbf{dom}(socks)) \wedge$
$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$

(* Assert that the socket meets some sanity properties. This is logically superfluous but aids semi-automatic model checking. See sane_socket (p36) for further details. *)
sane_socket $sock \wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$

(* The segment must be of an acceptable form *)
(* Note: some segment fields (namely TCP options *ws* and *mss*), are only used during connection establishment and any values assigned to them in segments during a connection are simply ignored. They are equal to the identifiers *ws_discard* and *mss_discard* respectively, which are otherwise unconstrained. *)
$iflgs.RST = \mathbf{F} \wedge$

(* The socket is fully connected so its complete address quad must match the address quad of the segment *seg*. By definition, *sock* is the socket with the best address match thus the auxiliary function tcp_socket_best_match is not required here. *)
$sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge$
$sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2 \wedge$

(* The socket must be in a connected state, or is in the $SYN\_RECEIVED$ state and *seg* is the final segment completing a passive or simultaneous open. *)
$tcp\_sock.st \notin \{CLOSED; LISTEN; SYN\_SENT\} \wedge$

$tcp\_sock.st \in \{SYN\_RECEIVED; ESTABLISHED; CLOSE\_WAIT; FIN\_WAIT\_1; FIN\_WAIT\_2;$
$CLOSING; LAST\_ACK; TIME\_WAIT\} \wedge$

(* 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? *)
$(\exists cond.\neg(tcp\_sock.st \in \{FIN\_WAIT\_1; CLOSING; LAST\_ACK; FIN\_WAIT\_2; TIME\_WAIT\} \wedge$
$cond)) \wedge$

(* A *SYN* should be received only in the *SYN_RECEIVED* state. *)
$(iflgs.SYN \implies tcp\_sock.st = SYN\_RECEIVED) \wedge$

(* 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 $ack \geq cb.snd\_max$. *)

$(ourfinisacked \implies wesentafin) \wedge$

(* *wercvdafin* approximated by *iflgs.FIN* *)
$(wercvdafin = iflgs.FIN) \wedge$

(* Process the segment and return an updated socket state *)

(

$\exists sock_0.$ di3_topstuff $sock\ sock_0 \wedge$

$\exists sock_1\ FIN_1\ stop_1.$ di3_ackstuff $tcp\_sock\ ourfinisacked\ h.arch\ h.rttab\ h.ifds\ sock_0(sock_1, FIN_1, stop_1) \wedge$
if $stop_1 = \mathbf{T}$
then
$(sock', oflgs.FIN) = (sock_1, FIN_1)$
else

let *datastuff theststuff* =
(* Extract and reassemble data (including urgent data). See di3_datastuff (p216). *)
di3_datastuff *wercvdafin theststuff ourfinisacked*
and *ststuff FIN_reass* =
(* Possibly change the socket's state (especially on receipt of a valid *FIN*). See di3_ststuff (p216). *)
di3_ststuff *wercvdafin ourfinisacked*
in
$\exists sock_2\ FIN_2.datastuff\ ststuff\ sock_1(sock_2, FIN_2) \wedge$
$(sock', oflgs.FIN) = (sock_2, FIN_2 \vee FIN_1)$
$) \wedge$

$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$
$sock'' = sock' \wedge$

(* 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 (p219) does all the hard work, updating the relevant sockets in the finite map *socks* to yield *socks'*. *)
(**if** $tcp\_sock.st = SYN\_RECEIVED \wedge$
$tcp\_sock'.st \in \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSE\_WAIT\}$ **then**

di3_socks_update $sid(socks \oplus (sid, sock''))socks'$
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. *)
$socks' = socks \oplus (sid, sock'')) \wedge$
write$(i_1, p_1, i_2, p_2)(oflgs, [\,])s'\ s''$

– *deliver_in_3* **initial checks :**
di3_topstuff $sock\ sock' =$
$\exists tcp\_sock.$
$sock.pr =$ TCP_PROTO $tcp\_sock \wedge$
let $cb = tcp\_sock.cb$ **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 (p51) for further details *)
choose tt_keep' :: update_idle tcp_sock.

$sock' = sock$ ⦇ $pr :=$ TCP_PROTO$(tcp_sock$
⦇ $cb := tcp\_sock.cb$ ⦇ $tt\_keep := tt\_keep'$⦈⦈)
⦈

– *deliver_in_3* **new ack processing, used in** di3_ackstuff **:**
di3_newackstuff $tcp\_sock\_0\ ourfinisacked\ arch\ rttab\ ifds\ sock(sock', FIN, sto\_p) =$

$\exists(sock'', FIN'', stop'') :: \{(sock', FIN, sto\_p) \mid$
$\exists t\_dupacks :: (UNIV : \mathsf{num\ set}).$
$\exists ack\_lt\_snd\_recover :: \{\mathbf{T}; \mathbf{F}\}.$
(**if** $\neg TCP\_DO\_NEWRENO \vee t\_dupacks < 3$ **then**
$(sock', FIN, sto\_p) = (sock, \mathbf{F}, \mathbf{F})$
else if $TCP\_DO\_NEWRENO \wedge t\_dupacks \geq 3 \wedge ack\_lt\_snd\_recover$ **then**

(* Attempt to create a segment for output using the modified control block (this is a relational monad idiom) *)
$stream\_mlift\_tcp\_output\_perhaps\_or\_fail\ arch\ rttab\ ifds\ sock(sock', FIN) \wedge$
$sto\_p = \mathbf{F}$

else if $TCP\_DO\_NEWRENO \wedge t\_dupacks \geq 3 \wedge \neg ack\_lt\_snd\_recover$ **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. *)
$(sock', FIN, sto\_p) = (sock, \mathbf{F}, \mathbf{F})$

else *ASSERTION_FAILURE*"di3_newackstuff" (* impossible *)

)}.
(* we never stop in the above, so always continue, but rebind sock *)
let $sock = sock''$ **in**
$\exists(sock''', FIN''', stop''') :: \{(sock', FIN, sto\_p) \mid$

(* 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". *)

$\exists IS\_SOME\_emission\_time :: \{\mathbf{T};\mathbf{F}\}.$
$\exists tcp\_sock.$
$(sock.pr = \mathrm{TCP\_PROTO}\ tcp\_sock) \wedge$
(* rebind sock in the process of updating *)
let $sock = sock \ \llparenthesis\ pr := \mathrm{TCP\_PROTO}(tcp\_sock \ \llparenthesis\ cb := tcp\_sock.cb$
$\llparenthesis$

(* 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. *)
$t\_softerror \ \hat{:=}\ *$ **onlywhen** $IS\_SOME\_emission\_time$
$\rrparenthesis \rrparenthesis) \rrparenthesis$ **in**

$\exists ack\_gt\_snd\_una :: \{\mathbf{T};\mathbf{F}\}.$
$\exists tcp\_sock.$
$(sock.pr = \mathrm{TCP\_PROTO}\ tcp\_sock) \wedge$
(**if** $tcp\_sock\_0.st = LAST\_ACK \wedge ourfinisacked$ **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 *)
$sock' = \mathrm{tcp\_close}\ arch\ sock \wedge$
$FIN = \mathbf{F} \wedge$
$sto\_p = \mathbf{T}$
else
(* Otherwise, flag that *deliver_in_3* can continue processing the segment if need be *)
$(sock', FIN, sto\_p) = (sock, \mathbf{F}, \mathbf{F}))$
}.
$sock' = sock''' \wedge$
$(FIN = (FIN'' \vee FIN''')) \wedge$
$sto\_p = stop'''$

– *deliver_in_3* **ACK processing :**
di3_ackstuff $tcp\_sock\_0\ ourfinisacked\ arch\ rttab\ ifds\ sock(sock', FIN, sto\_p) =$
$\exists ack\_le\_snd\_una :: \{\mathbf{T};\mathbf{F}\}.$
$\exists maybe\_dup\_ack :: \{\mathbf{T};\mathbf{F}\}.$
if $ack\_le\_snd\_una \wedge maybe\_dup\_ack$ **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. *)

$\exists t\_dupacks' :: (UNIV\ \mathbf{diff}\{0 : \mathsf{num}\}).$
$\exists ack\_lt\_snd\_recover :: \{\mathbf{T};\mathbf{F}\}.$

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. *)
$(sock', FIN, sto\_p) = (sock, \mathbf{F}, \mathbf{F})$

else if $t\_dupacks' > 3 \vee (t\_dupacks' = 3 \wedge TCP\_DO\_NEWRENO \wedge ack\_lt\_snd\_recover)$ **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-or-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". *)

$stream\_mlift\_tcp\_output\_perhaps\_or\_fail\ arch\ rttab\ ifds\ sock(sock', FIN) \wedge$

$sto\_p = \mathbf{T}$ (* no need to process the segment any further *)

else if $t\_dupacks' = 3 \wedge \neg(TCP\_DO\_NEWRENO \wedge ack\_lt\_snd\_recover)$ **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 *)

(* Attempt to create a segment for output using the modified control block (this is all a relational monad idiom) *)

$stream\_mlift\_tcp\_output\_perhaps\_or\_fail\ arch\ rttab\ ifds\ sock(sock', FIN) \wedge$

$sto\_p = \mathbf{T}$ (* no need to process the segment any further *)

else *ASSERTION_FAILURE*"di3_ackstuff: Believed to be impossible—here for completion and safety"

else if $ack\_le\_snd\_una \wedge \neg maybe\_dup\_ack$ **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. *)

$(sock', FIN, sto\_p) = (sock, \mathbf{F}, \mathbf{F})$

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 (p214) *)

di3_newackstuff $tcp\_sock\_0\ ourfinisacked\ arch\ rttab\ ifds\ sock(sock', FIN, sto\_p)$

– *deliver_in_3* **data processing :**

(di3_datastuff$(FIN_reass : \mathsf{bool})$ $the\_ststuff\ ourfinisacked\ sock(sock' : \mathsf{socket}, FIN : \mathsf{bool})) : \mathsf{bool} =$

let tcp_sock = tcp_sock_of $sock$ **in**

if $tcp\_sock.st = TIME\_WAIT \vee (tcp\_sock.st = CLOSING \wedge ourfinisacked)$ **then**
$the\_ststuff\ \mathbf{F}\ sock(sock', FIN)$
else
$the\_ststuff\ FIN\_reass\ sock(sock', FIN)$

– *deliver_in_3* **TCP state change processing :**

di3_ststuff $FIN\_reass\ ourfinisacked\ sock(sock', stop') =$

(* 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 *)

(* 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. *)

```
let sock = (if FIN_reass then
      sock ⦃ cantrcvmore := T⦄
   else sock) in
let tcp_sock = tcp_sock_of sock in
let cont = (sock' = sock ∧ stop' = F) in
let enter_TIME_WAIT = (sock' = sock
        ⦃ pr := TCP_PROTO(tcp_sock
          ⦃ st := TIME_WAIT;
            cb := tcp_sock.cb ⦃ tt_keep := ∗⦄
          ⦄)
        ⦄ ∧
  stop' = F) in
```

(* 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

    (* REMARK we are very loose here *)
  (SYN_RECEIVED, F) → (* In SYN_RECEIVED and have not received a FIN *)
    (∃ack_ge_suc_iss :: {T; F}.
    if ack_ge_suc_iss then
       (* This socket's initial SYN has been acknowledged *)
      sock' = sock ⦃ pr := TCP_PROTO(tcp_sock
                    ⦃ st := if ¬sock.cantsndmore then
                          ESTABLISHED (* socket is now fully connected *)
                       else
                          (* The connecting socket had it's 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
                    ⦄)⦄ ∧
      stop' = F
     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 *)
    sock' = sock ⦃ pr := TCP_PROTO(tcp_sock ⦃ st := CLOSE_WAIT⦄)⦄ ∧
    stop' = F ‖

  (ESTABLISHED, F) → (* In ESTABLISHED and have not received a FIN *)
     (* Doing common-case data delivery and acknowledgements. Remain in ESTABLISHED. *)
    cont ‖
```

$(ESTABLISHED, \mathbf{T}) \rightarrow$ (* In *ESTABLISHED* and received a *FIN* *)
(* Move into the *CLOSE_WAIT* state *)
$sock' = sock \ (\!| \ pr := \mathrm{TCP\_PROTO}(tcp\_sock \ (\!| \ st := CLOSE\_WAIT |\!))|\!) \ \wedge$
$stop' = \mathbf{F}$ ‖

$(CLOSE\_WAIT, \mathbf{F}) \rightarrow$ (* 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 ‖

$(CLOSE\_WAIT, \mathbf{T}) \rightarrow$ (* 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 ‖

$(FIN\_WAIT\_1, \mathbf{F}) \rightarrow$ (* 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. *)
$sock' = sock \ (\!| \ pr := \mathrm{TCP\_PROTO}(tcp\_sock \ (\!| \ st := FIN\_WAIT\_2 |\!))|\!) \ \wedge$
$stop' = \mathbf{F}$

else
(* If this socket's *FIN* has not been acknowledged then remain in *FIN_WAIT_1* *)
cont ‖

$(FIN\_WAIT\_1, \mathbf{T}) \rightarrow$ (* 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 *)
$sock' = sock \ (\!| \ pr := \mathrm{TCP\_PROTO}(tcp\_sock \ (\!| \ st := CLOSING |\!))|\!) \ \wedge$
$stop' = \mathbf{F}$ ‖

$(FIN\_WAIT\_2, \mathbf{F}) \rightarrow$ (* 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 ‖

$(FIN\_WAIT\_2, \mathbf{T}) \rightarrow$ (* In *FIN_WAIT_2* and have received a *FIN* *)
(* Connection has been shutdown so enter *TIME_WAIT* *)
enter_TIME_WAIT ‖

$(CLOSING, \mathbf{F}) \rightarrow$ (* 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 previosuly emitted *FIN* is not acknowledged this socket's retransmit timer will eventually fire causing retransmission of the *FIN*. *)
cont ∥

$(CLOSING, \mathbf{T}) \rightarrow$ (* 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 ∥

$(LAST\_ACK, \mathbf{F}) \rightarrow$ (* 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 ∥

$(LAST\_ACK, \mathbf{T}) \rightarrow$ (* 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 *)
ASSERTION_FAILURE"di3_ststuff" (* impossible *) ∥

$(TIME\_WAIT, \mathbf{F}) \rightarrow$ (* In *TIME_WAIT* and have not received a *FIN* *)
(* Remaining in *TIME_WAIT* until the `2MSL` timer expires *)
cont ∥

$(TIME\_WAIT, \mathbf{T}) \rightarrow$ (* 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 establishement 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* (p212) for further details. *)

let *interesting* = $\lambda sid'$.
$sid' \neq sid \wedge$
case (*socks*[sid']).*pr* **of**
UDP_PROTO *udp_sock* → **F**
∥ TCP_PROTO(tcp_sock') →
case tcp_sock'.*lis* **of**
$* \rightarrow$ **F**
∥ ↑ *lis* →

$sid \in lis.q0$ **in**

let $interesting\_sids = (\mathbf{dom}(socks)) \cap interesting$ **in**

if $interesting\_sids \neq \{\}$ **then**
(* There exists another socket $sock'$ that is listening and has socket $sock\_1$ referenced by sid on its queue of incomplete connections $lis.q0$. *)
$\exists sid'\ sock'\ tcp\_sock'\ lis\ q0L\ q0R.$
$sid' \in interesting\_sids \wedge$
$sock' = socks[sid'] \wedge$
$sock'.pr = \text{TCP\_PROTO}\ tcp\_sock' \wedge$
$sid' \neq sid \wedge$
$tcp\_sock'.lis = \uparrow lis \wedge$
$lis.q0 = q0L\ @\ (sid :: q0R) \wedge$

(* 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 $lis' = lis\ \langle\!\langle\ q0 := q0L\ @\ q0R;$
$q := sid :: lis.q \rangle\!\rangle$ **in**
let $cb' = tcp\_sock\_1.cb$ **in**

(* Update both the newly connected socket and the listening socket *)
$socks' = socks \oplus$
$[(sid, sock\_1\ \langle\!\langle\ pr := \text{TCP\_PROTO}(tcp\_sock\_1\ \langle\!\langle\ cb := cb' \rangle\!\rangle) \rangle\!\rangle);$
$(sid', sock'\ \langle\!\langle\ pr := \text{TCP\_PROTO}(tcp\_sock'\ \langle\!\langle\ lis := \uparrow lis' \rangle\!\rangle) \rangle\!\rangle)]$
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 $lis' = lis\ \langle\!\langle\ q0 := q0L\ @\ q0R \rangle\!\rangle$ **in**
$socks' = socks \oplus (sid', sock'\ \langle\!\langle\ pr := \text{TCP\_PROTO}(tcp\_sock'\ \langle\!\langle\ lis := \uparrow lis' \rangle\!\rangle) \rangle\!\rangle)$

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. *)
$socks' = socks$

---

deliver_in_3b **tcp: network nonurgent** **Receive data after process has gone away**

$(h\ \langle\!\langle socks := socks;$ $\xrightarrow{\tau}$ $(h\ \langle\!\langle socks := socks';$
$iq := iq;$ $\quad iq := iq';$
$oq := oq;$ $\quad oq := oq';$
$bndlm := bndlm \rangle\!\rangle,$ $\quad bndlm := bndlm' \rangle\!\rangle,$
$SS, MM)$ $\quad SS', MM)$

(* **Summary:** if data arrives after the process associated with a socket has gone away, close socket and emit RST segment. *)

$sid \in \mathbf{dom}(socks) \wedge$

$sock\_0 = socks[sid] \wedge$
$sock\_0.is_1 = \uparrow i_1 \wedge sock\_0.ps_1 = \uparrow p_1 \wedge sock\_0.is_2 = \uparrow i_2 \wedge sock\_0.ps_2 = \uparrow p_2 \wedge$
$sock\_0.pr = \mathrm{TCP\_PROTO}(tcp\_sock\_0) \wedge$

$\exists S_0\ s\ s'.SS = S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\mathrm{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$iflgs = iflgs\ \langle\!\langle\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{F} \rangle\!\rangle \wedge$
$idata\ \in\ UNIV \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

(* test that this is data arriving after process has gone away *)
$tcp\_sock\_0.st\ \in \{FIN\_WAIT\_1; CLOSING; LAST\_ACK; FIN\_WAIT\_2; TIME\_WAIT\} \wedge$
$sock\_0.fid = * \wedge$

(* close socket and emit RST segment *)
$socks' = socks \oplus (sid, \mathrm{tcp\_close}\ \ h.arch\ sock\_0) \wedge$
$oflgs = oflgs\ \langle\!\langle\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{T} \rangle\!\rangle \wedge$
$odata\ \in\ UNIV \wedge$
$\exists s''.$
$\mathrm{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s'\ s'' \wedge$
$\mathrm{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')])SS'$

---

deliver_in_4 **tcp: network nonurgent** **Receive and drop (silently) a non-sane or martian segment**

$(h\ \langle\!\langle iq := iq \rangle\!\rangle, SS, MM)\ \ \xrightarrow{\tau}\ \ (h\ \langle\!\langle iq := iq' \rangle\!\rangle, SS, MM)$

(* **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**. *)

$dequeue\_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$
$seg.is_2 = \uparrow i_2 \wedge$
$is_1 = seg.is_1 \wedge$
$i_2\ \in\ local\_ips(h.ifds) \wedge$
$(\mathbf{F} \vee$ (* placeholder for segment checksum and offset field not sensible *)
$\neg($
  $\mathbf{T} \wedge$ (* placeholder for not a link-layer multicast or broadcast *)
  $\neg(is\_broadormulticast\ h.ifds\ i_2) \wedge$ (* seems unlikely, since $i_1\ \in\ local\_ips\ h.ifds$ *)
  $\neg(is_1 = *) \wedge$
  $\neg is\_broadormulticast\ h.ifds(\mathbf{the}\ \ is_1)$
$)$
$)$

---

deliver_in_5 **tcp: network nonurgent** **Receive and drop (maybe with RST) a sane segment**

that does not match any socket

$(h \; \langle\!| iq := iq;$ $\xrightarrow{\tau}$ $(h \; \langle\!| iq := iq';$
$oq := oq;$ $oq := oq';$
$bndlm := bndlm |\!\rangle,$ $bndlm := bndlm' |\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$ $SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$

(* **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'. *)

$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\;\mathbf{F}(iflgs, idata)s\; s' \land$

$i_1 \;\in\; local\_ips(h.ifds) \land$

$\mathbf{T}\land$ (* placeholder for segment checksum and offset field sensible *)

$\neg(\exists((sid, sock) :: h.socks)tcp\_sock.$
$\quad sock.pr = \text{TCP\_PROTO}(tcp\_sock) \land$
$\quad \text{match\_score}(sock.is_1, sock.ps_1, sock.is_2, sock.ps_2)$
$\qquad\qquad (i_2, \uparrow p_2, i_1, \uparrow p_1) > 0$
$\;) \land$

$\text{dropwithreset}\;\; iflgs.RST(\uparrow i_2, \uparrow i_1)h.ifds\;\; oflgs.RST \land$
$oflgs.SYN = \mathbf{F} \land$
$oflgs.SYNACK = \mathbf{F} \land$
$oflgs.FIN = \mathbf{F} \land$
$odata = [\,] \land$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s'\; s''$

---

deliver_in_7 **tcp: network nonurgent** **Receive RST and zap non-**$\{CLOSED$**;** $LISTEN$**;** SYN_SENT**;** $SYN\_RECEIVED$**;** $TIME\_WAIT\}$ **socket**

$(h \; \langle\!| ts := ts \oplus (tid \mapsto (ts_{st})_d);$ $\xrightarrow{\tau}$ $(h \; \langle\!| ts := ts \oplus (tid \mapsto (ts_{st})_d);$
$socks := socks \oplus [(sid, sock)];$ $socks := socks \oplus [(sid, sock')];$
$iq := iq |\!\rangle,$ $iq := iq' |\!\rangle,$
$SS, MM)$ $SS', MM)$

(* **Summary:** receive RST and silently zap non-$\{CLOSED$; $LISTEN$; SYN_SENT; $SYN\_RECEIVED$; $TIME\_WAIT\}$ socket *)

$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\qquad\qquad \text{TCP\_Sock}(st, cb, *)) \land$
$st \;\notin \{CLOSED; LISTEN; SYN\_SENT; SYN\_RECEIVED; TIME\_WAIT\} \land$

$\exists S_0\; s\; s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \land$
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\;\mathbf{F}(iflgs, idata)s\; s' \land$
$iflgs.RST = \mathbf{T} \land$
$idata \;\in\; UNIV \land$

((* $sock.st \;\in \{CLOSED; LISTEN; SYN\_SENT; SYN\_RECEIVED; TIME\_WAIT\}$ excluded already above *)
if $st \;\in \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSE\_WAIT\}$ **then**
$\quad err = \uparrow ECONNRESET$
else (* $sock.st \;\in \{CLOSING; LAST\_ACK\}$ – leave existing error *)

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

$err = sock.es) \wedge$

(* see tcp_close (p52) *)
$sock' = \text{tcp\_close}\ \ h.arch(sock\ \llparenthesis\ es := err\rrparenthesis) \wedge$
$\text{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')])SS'$

deliver_in_7a **tcp: network nonurgent** **Receive RST and zap** *SYN_RECEIVED* **socket**

$(h\ \llparenthesis socks := socks \oplus [(sid, sock)];\ iq := iq\rrparenthesis,\ SS, MM) \xrightarrow{\tau} (h\ \llparenthesis socks := socks \oplus socks\_update';\ iq := iq'\rrparenthesis,\ SS', MM)$

(* **Summary:** receive RST and zap *SYN_RECEIVED* socket, removing from listen queue etc. *)

$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$iflgs.RST = \mathbf{T} \wedge$
$idata\ \in\ UNIV \wedge$

$sid\ \notin \mathbf{dom}(socks) \wedge$

$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(SYN\_RECEIVED, cb, *)) \wedge$

((* There is a corresponding listening socket – passive open *)
$(\exists(sid', lsock) :: socks \backslash\backslash sid.$
$\exists tcp\_lsock\ lis\ q0L\ q0R\ lsock'.$
$\quad lsock.pr = \text{TCP\_PROTO}(tcp\_lsock) \wedge$
$\quad tcp\_lsock.st = LISTEN \wedge$
$\quad tcp\_lsock.lis = \uparrow lis \wedge$
$\quad lis.q0 = q0L \,@\, (sid :: q0R) \wedge$
$\quad lsock' = lsock$
$\quad\quad \llparenthesis\ pr := \text{TCP\_PROTO}(tcp\_lsock\ \llparenthesis\ lis :=$
$\quad\quad\quad \uparrow(lis\ \llparenthesis\ q0 := q0L \,@\, q0R\rrparenthesis)\rrparenthesis)\rrparenthesis \wedge$
$\quad socks\_update' = [(sid', lsock'); (sid, sock')]$
$) \vee$
((* 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 *)
$sock' = (\text{tcp\_close}\ \ h.arch\ sock)\llparenthesis\ ps_1 := \mathbf{if}\ \ bsd\_arch\ h.arch\ \ \mathbf{then}\ \ *\ \ \mathbf{else}\ \ sock.ps_1\rrparenthesis \wedge$
$\text{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')])SS'$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

deliver_in_7b **tcp: network nonurgent** **Receive RST and ignore for *LISTEN* socket**

$(h\ \langle\!|socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $(h\ \langle\!|socks := socks \oplus [(sid, sock)];$
$iq := iq|\!\rangle,$ $iq := iq'|\!\rangle,$
$SS, MM)$ $SS, MM)$

(* **Summary:** receive RST and ignore for *LISTEN* socket *)

$dequeue\_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(LISTEN, cb, lis)) \wedge$

(* 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 $i_1 \in local\_ips\ h.ifds$ *)
$\neg(is\_broadormulticast\ h.ifds\ i_1) \wedge$
$\neg(is\_broadormulticast\ h.ifds\ i_2) \wedge$
$(\textbf{case}\ is_1\ \textbf{of}$
$\quad \uparrow i_1' \rightarrow i_1' = i_1\ \|$
$\quad * \rightarrow \mathbf{T}) \wedge$

$(\exists seq\_discard\ ack\_discard\ URG\_discard\ ACK\_discard\ PSH\_discard\ SYN\_discard\ FIN\_discard$
$\quad win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard.$
$seg = \langle\!|$
$\quad is_1 := \uparrow i_2;$
$\quad is_2 := \uparrow i_1;$
$\quad ps_1 := \uparrow p_2;$
$\quad ps_2 := \uparrow p_1;$
$\quad seq := tcp\_seq\_flip\_sense(seq\_discard : tcp\_seq\_foreign);$
$\quad ack := tcp\_seq\_flip\_sense(ack\_discard : tcp\_seq\_local);$
$\quad URG := URG\_discard;$
$\quad ACK := ACK\_discard;$
$\quad PSH := PSH\_discard;$
$\quad RST := \mathbf{T};$
$\quad SYN := SYN\_discard;$
$\quad FIN := FIN\_discard;$
$\quad win := win\_discard;$
$\quad ws := ws\_discard;$
$\quad urp := urp\_discard;$
$\quad mss := mss\_discard;$
$\quad ts := ts\_discard;$
$\quad data := data\_discard$
$\quad |\!\rangle$
$) \wedge$

$\text{tcp\_socket\_best\_match}(socks \backslash\backslash sid)(sid, sock) seg\ h.arch$ (* there does not exist a better socket match to which the segment should be sent *)

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

deliver_in_7c **tcp: network nonurgent** **Receive RST and ignore for** *SYN_SENT* **(unacceptable ack) or** *TIME_WAIT* **socket**

$(h\ \langle\!\lbrack socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $(h\ \langle\!\lbrack socks := socks \oplus [(sid, sock')];$
$iq := iq\rbrack\!\rangle,$ $iq := iq'\rbrack\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$ $SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$

(* **Summary:** receive RST and ignore for *SYN_SENT* (unacceptable ack) or *TIME_WAIT* socket *)

$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$sid \notin \mathbf{dom}(socks) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *)) \wedge$
$st \in \{SYN\_SENT; TIME\_WAIT\} \wedge$

$iflgs.RST = \mathbf{T} \wedge$
$idata \in UNIV \wedge$

(* no- or unacceptable- ACK *)
$(st = SYN\_SENT \implies \mathbf{F}) \wedge$

$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
(**if** $st = TIME\_WAIT$ **then** (* only update if $\geq$ *ESTABLISHED*, c.f. `tcp\_input.c:887` *)
$\quad sock' = sock\ \langle\!\lbrack\ pr := \text{TCP\_PROTO}(tcp\_sock$
$\quad\quad \langle\!\lbrack\ cb := cb$
$\quad\quad\quad \langle\!\lbrack\ tt\_keep := \uparrow((())_{slow\_timer\ TCPTV\_KEEP\_IDLE})\rbrack\!\rangle$
$\quad\quad \rbrack\!\rangle)\rbrack\!\rangle$
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 *)
$\quad sock' = sock)$

---

deliver_in_7d **tcp: network nonurgent** **Receive RST and zap** *SYN_SENT* **(acceptable ack) socket**

$(h\ \langle\!\lbrack socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $(h\ \langle\!\lbrack socks := socks \oplus [(sid, sock')];$
$iq := iq\rbrack\!\rangle,$ $iq := iq'\rbrack\!\rangle,$
$SS, MM)$ $SS', MM)$

(* **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. *)

$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{F}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$sid \notin \mathbf{dom}(socks) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(SYN\_SENT, cb, *)) \wedge$

$iflgs.RST = \mathbf{T} \wedge$
$idata \in UNIV \wedge$

if *windows_arch h.arch* **then**

$sock' = sock$ (* Windows XP just ignores RST's with a valid ack during connection establishment *) $\wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$
else
$(\exists err.$
$err \in \{ECONNREFUSED; ECONNRESET\} \wedge$ (* Note it is unclear whether or not this error will overwrite any existing error on the socket *)
$sock' = (\text{tcp\_close}\ h.arch\ sock)\langle\!| ps_1 := \textbf{if}\ bsd\_arch\ h.arch\ \textbf{then}\ *\ \textbf{else}\ sock.ps_1; es := \uparrow err |\!\rangle \wedge$
$\text{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')])SS')$

---

deliver_in_8 **tcp: network nonurgent** **Receive SYN in non-{*CLOSED*; *LISTEN*; *SYN_SENT*; *TIME_WAIT*} state**

$$(h\ \langle\!|socks := socks \oplus [(sid, sock)]; iq := iq; oq := oq; bndlm := bndlm|\!\rangle, SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM) \xrightarrow{\tau} (h\ \langle\!|socks := socks \oplus [(sid, sock')]; iq := iq'; oq := oq'; bndlm := bndlm'|\!\rangle, SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$$

(* **Summary:** Receive a SYN in non-{*CLOSED*; *LISTEN*; *SYN_SENT*; *TIME_WAIT*} state. Drop it and (depending on the architecture) generate a RST. *)

$sid \notin \textbf{dom}(socks) \wedge$
$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *)) \wedge$
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$iflgs.RST = \mathbf{F} \wedge (iflgs.SYN \vee iflgs.SYNACK) \wedge$
$idata \in UNIV \wedge$

(* 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. *)

$st \notin \{CLOSED; LISTEN; SYN\_SENT; TIME\_WAIT\} \wedge$

$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\textbf{let}\ tt\_keep' = \textbf{if}\ tcp\_sock.st \neq SYN\_RECEIVED\ \textbf{then}$
$\quad \uparrow((())_{slow\_timer\ TCPTV\_KEEP\_IDLE})$
$\quad \textbf{else}$
$\quad tcp\_sock.cb.tt\_keep\ \textbf{in}$

$sock' = sock\ \langle\!|\ pr := \text{TCP\_PROTO}(tcp\_sock$
$\quad \langle\!|\ cb := tcp\_sock.cb\ \langle\!|\ tt\_keep := tt\_keep'|\!\rangle$
$\quad |\!\rangle)|\!\rangle \wedge$

$oflgs = oflgs\ \langle\!|\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{T}|\!\rangle \wedge$
$odata \in UNIV \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s'\ s''$

---

deliver_in_9 **tcp: network nonurgent** **Receive SYN in *TIME_WAIT* state if there is no match-**

ing *LISTEN* socket or sequence number has not increased

$(h\ \langle\!|socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $(h\ \langle\!|socks := socks \oplus [(sid, sock)];$
$iq := iq;$ $iq := iq';$
$oq := oq;$ $oq := oq';$
$bndlm := bndlm|\!\rangle,$ $bndlm := bndlm'|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$ $SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s'')], MM)$

(* **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. *)

$dequeue\_iq(iq, iq', \uparrow(TCP\ seg)) \wedge$

$sid \notin \mathbf{dom}(socks) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(TIME\_WAIT, cb, *)) \wedge$
$\text{read}(i_1, p_1, i_2, p_2)\mathbf{T}\ \mathbf{F}(iflgs, idata)s\ s' \wedge$
$iflgs.RST = \mathbf{F} \wedge (iflgs.SYN \vee iflgs.SYNACK) \wedge$
$idata \in UNIV \wedge$

(* no matching LISTEN socket, or the sequence number has not increased *)
$(\mathbf{T}$
$\vee$
$\neg(\exists((sid, sock) :: socks)tcp\_sock.$
$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.st = LISTEN \wedge$
$sock.is_1 \in \{*; \uparrow i_1\} \wedge$
$sock.ps_1 = \uparrow p_1)$
$) \wedge$

$oflgs = oflgs\ \langle\!|\ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{T}|\!\rangle \wedge$
$odata \in UNIV \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s'\ s''$

(* 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 9

Host LTS: TCP Output

9.1 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 (p45) and tcp_output_really (p45). 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 occuring 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_flgs_data (p262) and make_syn_ack_flgs_data (p263) 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 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 (p45) or the wrapper functions tcp_output_perhaps (p48) and *stream_mlift_tcp_output_perhaps_or_fail* (p50) 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 (p48) and *stream_mlift_tcp_output_perhaps_or_fail* (p50) 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 (p50), *stream_enqueue_or_fail_sock* (p50), enqueue_and_ignore_fail (p50), enqueue_each_and_ignore_fail (p50) or *stream_mlift_tcp_output_perhaps_or_fail* (p50). 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 (p48) 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.

9.1.1 Summary

deliver_out_1	**tcp: network nonurgent**	Common case TCP output

9.1.2 Rules

deliver_out_1 **tcp: network nonurgent** **Common case TCP output**

$(h\ \langle\!|socks := socks \oplus [(sid, sock)];$
$oq := oq|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$

$\xrightarrow{\tau}$

$(h\ \langle\!|socks := socks \oplus [(sid, sock'')];$
$oq := oq'|\!\rangle,$
$SS \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$

(* **Summary:** output TCP segment if possible. In some cases update the socket's persist timer without performing output. *)

(* The TCP socket is connected *)
$sid \notin \mathbf{dom}(socks) \wedge$
$sock = \text{SOCK}(fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore,$
$\quad cantrcvmore, \text{TCP\_PROTO}(tcp\_sock)) \wedge$
$tcp\_sock = \text{TCP\_Sock0}(st, cb, *) \wedge$

(* and either is in a synchronised state with initial *SYN* acknowledged... *)
$((st \in \{ESTABLISHED; CLOSE\_WAIT; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING;$
$\quad LAST\_ACK; TIME\_WAIT\}) \vee$
(* ...or is in the *SYN_SENT* or *SYN_RECEIVED* state and a *FIN* needs to be emitted *)
$(st \in \{SYN\_SENT; SYN\_RECEIVED\} \wedge cantsndmore)$
$) \wedge$

(* 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. *)
$(do\_output, persist\_fun) \in \text{tcp\_output\_required} \wedge$
$(do\_output \vee persist\_fun \neq *) \wedge$

(* Apply any persist timer side-effect from tcp_output_required *)
let $sock_0$ = **option_case** $sock$ $(\lambda f.sock$ ⦃ $pr :=$ TCP_PROTO(tcp_sock cb $:\hat{=}$ f)⦄) $persist\_fun$ **in**

(**if** do_output **then** (* output a segment *)
(* Construct the segment to emit, updating the socket's state *)
$stream\_tcp\_output\_really\ sock_0(sock', FIN) \wedge$

$sock'.pr =$ TCP_PROTO$(tcp_sock') \wedge$

(* Add the segment to the host's output queue, rolling back the socket's control block state if an error occurs *)
$oflgs =$⦃ $SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := FIN; RST := \mathbf{F}$⦄ $\wedge$
$odata = [\,] \wedge$
write$(i_1, p_1, i_2, p_2)(oflgs, odata) s\ s' \wedge$
$stream\_enqueue\_or\_fail\_sock(tcp\_sock'.st \in \{CLOSED; LISTEN; SYN\_SENT\}) h.arch\ h.rttab\ h.ifds$
$(\uparrow i_1, \uparrow i_2) sock_0\ sock'\ sock''$

else (* Do not output a segment, but ensure things are tidied up *)
$oq = oq' \wedge$
$sock'' = sock_0 \wedge$
$s' = s$
)

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

Chapter 10

Host LTS: TCP Timers

10.1 Timers (TCP only)

10.1.1 Summary

timer_tt_rexmtsyn_1	**tcp: misc nonurgent**	SYN retransmit timer expires
timer_tt_rexmt_1	**tcp: misc nonurgent**	retransmit timer expires
timer_tt_persist_1	**tcp: misc nonurgent**	persist timer expires
timer_tt_keep_1	**tcp: network nonurgent**	keepalive timer expires
timer_tt_2msl_1	**tcp: misc nonurgent**	2*MSL timer expires
timer_tt_delack_1	**tcp: misc nonurgent**	delayed-ACK timer expires
timer_tt_conn_est_1	**tcp: misc nonurgent**	connection establishment timer expires
timer_tt_fin_wait_2_1	**tcp: misc nonurgent**	*FIN_WAIT_2* timer expires

10.1.2 Rules

timer_tt_rexmtsyn_1 **tcp: misc nonurgent** **SYN retransmit timer expires**

$(h\ \langle\!|socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $(h\ \langle\!|socks := socks \oplus [(sid, sock')];$
$oq := oq|\!\rangle,$ $oq := oq'|\!\rangle,$
$SS, MM)$ $SS', MM)$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$shift \in UNIV \wedge$
$tcp\_sock.st = SYN\_SENT \wedge$ (* this rule is incomplete: *RexmtSyn* is possible in other states, since *deliver_in_2* may change state without clearing *tt_rexmt* *)

$cb = tcp\_sock.cb \wedge$

$\exists i_1\ i_2\ p_1\ p_2.(sock.is_1, sock.is_2, sock.ps_1, sock.ps_2) = (\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2) \wedge$
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 *)
$\mathrm{tcp\_drop\_and\_close}\ h.arch(\uparrow ETIMEDOUT) sock(sock', (oflgs, odata)) \wedge$
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\mathrm{write}(i_1, p_1, i_2, p_2)(oflgs, odata) s\ s' \wedge$
$\mathrm{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]) SS'$

else
(* Update the control block based upon the number of occasions on which the timer expired *)

$cb' = cb \wedge$

$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
(* Create the segment to be retransmitted *)
$(\mathit{oflgs}, \mathit{odata}) \in$ make_syn_flgs_data $\wedge$
$\text{write}(i_1, p_1, i_2, p_2)(\mathit{oflgs}, \mathit{odata})s\ s' \wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')] \wedge$
(* Attempt to add the new segment to the host's output queue, constraining the final control block state *)
$\mathit{stream\_enqueue\_or\_fail}\ \mathbf{F}\ h.\mathit{arch}\ h.\mathit{rttab}\ h.\mathit{ifds}(\uparrow i_1, \uparrow i_2)cb'\ cb'' \wedge$
$\mathit{sock}' = \mathit{sock}\ \langle\!| \mathit{pr} := \text{TCP\_PROTO}(\mathit{tcp\_sock}\ \langle\!| \mathit{cb} := cb'' |\!\rangle)|\!\rangle$

---

timer_tt_rexmt_1 **tcp: misc nonurgent** **retransmit timer expires**

$(h\ \langle\!|\mathit{socks} := \mathit{socks} \oplus [(\mathit{sid}, \mathit{sock})]; \mathit{oq} := \mathit{oq}|\!\rangle, SS, MM) \xrightarrow{\tau} (h\ \langle\!|\mathit{socks} := \mathit{socks} \oplus [(\mathit{sid}, \mathit{sock}'')]; \mathit{oq} := \mathit{oq}'|\!\rangle, SS', MM)$

$\mathit{sock}.\mathit{pr} = \text{TCP\_PROTO}(\mathit{tcp\_sock}) \wedge$
$\mathit{sock}'.\mathit{pr} = \text{TCP\_PROTO}(\mathit{tcp\_sock}') \wedge$
$(\mathit{tcp\_sock}.\mathit{st} \notin \{\mathit{CLOSED}; \mathit{LISTEN}; \mathit{SYN\_SENT}; \mathit{CLOSE\_WAIT}; \mathit{FIN\_WAIT\_2}; \mathit{TIME\_WAIT}\} \vee$
$(\mathit{tcp\_sock}.\mathit{st} = \mathit{LISTEN} \wedge \mathit{bsd\_arch}\ h.\mathit{arch})) \wedge$

$\mathit{shift} \in \mathit{UNIV} \wedge$

$cb = \mathit{tcp\_sock}.cb \wedge$

(**if**
$\mathit{shift} + 1 >$ (**if** $\mathit{tcp\_sock}.\mathit{st} = \mathit{SYN\_RECEIVED}$ **then**
$\mathit{TCP\_SYNACKMAXRXTSHIFT}$ **else** $\mathit{TCP\_MAXRXTSHIFT}$)
then
(* Note that BSD's syncaches have a much lower threshold for retransmitting SYN,ACKs than normal *)
(* drop connection *)

tcp_drop_and_close $h.\mathit{arch}(\uparrow \mathit{ETIMEDOUT})\mathit{sock}(\mathit{sock}', (\mathit{oflgs}, \mathit{odata})) \wedge$ (* will always get exactly one segment *)
if exists_quad_of sock **then**
let $(i_1, p_1, i_2, p_2) =$ quad_of sock **in**
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(\mathit{oflgs}, \mathit{odata})s\ s' \wedge$
case $\mathit{tcp\_sock}.\mathit{st} = \mathit{LISTEN}$ **of**
$\mathbf{T} \rightarrow SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$
$\|\ \mathbf{F} \rightarrow \text{destroy}(i_1, p_1, i_2, p_2)(S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')])SS'$
else
$SS' = SS$

else

(* backoff the timer and do a retransmit *)
$cb' = cb \wedge$

(**if** $\mathit{tcp\_sock}.\mathit{st} = \mathit{SYN\_RECEIVED}$ **then**
$(\exists i_1\ i_2\ p_1\ p_2.$

(* 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_tstmp* is **F** due to the option negotiation in *deliver_in_1*. Then *tf_doing_tstmp* is necessarily **F** too and the retransmitted SYN,ACK segment does not contain a timestamp. OTOH, if *tf_req_tstmp* is still **T** then so is *tf_doing_tstmp* 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 $cb''' = cb'$ **in**

(* Note that *tt_delack* and possibly other timers should be cleared here *)
$(sock.is_1, sock.is_2, sock.ps_1, sock.ps_2) = (\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2) \wedge$

(* 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_advmss* to set the same mss option in the segment, so we do not need to do anything special here. *)
$oflgs = \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{T}; FIN := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge$
$odata = [\,] \wedge$
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')] \wedge$

(* 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 *)
$sock' = sock\ \langle\!| pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!| cb := cb' |\!\rangle) |\!\rangle$
)
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. *)
$bsd\_arch\ h.arch \wedge$
$(\exists i_1\ i_2\ p_1\ p_2.$
$(sock.is_1, sock.is_2, sock.ps_1, sock.ps_2) = (\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2) \wedge$
$sock.cantsndmore \implies oflgs.FIN \wedge$
$oflgs = oflgs\ \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge$
$odata = [\,] \wedge$
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')] \wedge$
(* Retransmission only continues if *FIN* is set in the outgoing segment (really!) *)
$sock' = sock\ \langle\!| pr := \text{TCP\_PROTO}(tcp\_sock$
$\langle\!| cb := cb' |\!\rangle) |\!\rangle)$
else (* *ESTABLISHED*, *FIN_WAIT_1*, *CLOSING*, *LAST_ACK* *)
(* i.e., cannot be *CLOSED*, *LISTEN*, *SYN_SENT*, *CLOSE_WAIT*, *FIN_WAIT_2*, *TIME_WAIT* *)
stream_tcp_output_really
$(sock\ \langle\!| pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!| cb := cb' |\!\rangle) |\!\rangle)$
$(sock', oflgs.FIN)$ (* always emits exactly one segment *) $\wedge$
$oflgs = oflgs\ \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge$
$odata = [\,] \wedge$
let $(i_1, p_1, i_2, p_2) = \text{quad\_of}\ sock$ **in**
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

)

) $\wedge$

$stream\_enqueue\_or\_fail\ \mathbf{T}\ h.arch\ h.rttab\ h.ifds(sock'.is_1, sock'.is_2) tcp\_sock'.cb\ cb'' \wedge$
$sock'' = sock' \langle\!| pr := \mathrm{TCP\_PROTO}(tcp\_sock' \langle\!| cb := cb'' |\!\rangle) |\!\rangle$

---

timer_tt_persist_1 **tcp: misc nonurgent** **persist timer expires**

$(h \langle\!| socks := socks \oplus [(sid, sock)]; oq := oq |\!\rangle, SS \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM)$ $\xrightarrow{\tau}$ $(h \langle\!| socks := socks \oplus [(sid, sock'')]; oq := oq' |\!\rangle, SS \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$
let $sock_0 = sock$ **in**
$stream\_tcp\_output\_really$
$sock_0$
$(sock', oflgs.FIN) \wedge$
$oflgs = oflgs \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge$
$odata = [\,] \wedge$
(* guaranteed by *stream_tcp_output_really* *)
$(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2) = (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2) \wedge$
$\mathrm{write}(i_1, p_1, i_2, p_2)(oflgs, odata) s\ s' \wedge$

$stream\_enqueue\_or\_fail\_sock(tcp\_sock'.st \in \{CLOSED; LISTEN; SYN\_SENT\}) h.arch\ h.rttab\ h.ifds$
$(\uparrow i_1, \uparrow i_2) sock_0\ sock'\ sock''$

---

timer_tt_keep_1 **tcp: network nonurgent** **keepalive timer expires**

$(h \langle\!| socks := socks \oplus [(sid, \mathrm{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \mathrm{TCP\_Sock}(st, cb, *)))]; oq := oq |\!\rangle, SS, MM)$

$\xrightarrow{\tau}$ $(h \langle\!| socks := socks \oplus [(sid, \mathrm{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \mathrm{TCP\_Sock}(st, cb', *)))]; oq := oq' |\!\rangle, SS, MM)$

(* 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 *)

$cb.tt\_keep = \uparrow((())_d) \wedge$
$timer\_expires\ d \wedge$

$cb' = cb \langle\!| tt\_keep := \uparrow((())_{slow\_timer\ TCPTV\_KEEPINTVL}) |\!\rangle$

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

timer_tt_2msl_1 **tcp: misc nonurgent** **2*MSL timer expires**

$(h \langle\!|socks := socks \oplus [(sid, sock)]|\!\rangle, SS, MM) \xrightarrow{\tau} (h \langle\!|socks := socks \oplus [(sid, sock')]|\!\rangle, SS', MM)$

(* **Summary:** When the 2MSL *TIME_WAIT* period expires, the socket is closed. *)

if exists_quad_of *sock* **then**
destroy(quad_of *sock*)SS SS'
else

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock' = \mathrm{tcp\_close}\ h.arch\ sock \wedge$
$SS' = SS$

timer_tt_delack_1 **tcp: misc nonurgent** **delayed-ACK timer expires**

$(h \langle\!|socks := socks \oplus [(sid, sock)]; oq := oq|\!\rangle, SS \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)], MM) \xrightarrow{\tau} (h \langle\!|socks := socks \oplus [(sid, sock'')]; oq := oq'|\!\rangle, SS \oplus [(\mathrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')], MM)$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$
let $sock_0 = sock$ **in**
$stream\_tcp\_output\_really\ sock_0(sock', oflgs.FIN) \wedge$
$oflgs = oflgs \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{F}; RST := \mathbf{F}|\!\rangle \wedge$
$odata = [\,] \wedge$
$(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2) = (sock_0.is_1, sock_0.ps_1, sock_0.is_2, sock_0.ps_2) \wedge$
$\mathrm{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
$stream\_enqueue\_or\_fail\_sock(tcp\_sock'.st \in \{CLOSED; LISTEN; SYN\_SENT\})h.arch\ h.rttab\ h.ifds$
$(sock_0.is_1, sock_0.is_2)sock_0\ sock'\ sock''$

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**

$(h \langle\!|socks := socks \oplus [(sid, sock)]; oq := oq|\!\rangle, SS, MM) \xrightarrow{\tau} (h \langle\!|socks := socks \oplus [(sid, sock')]; oq := oq'|\!\rangle, SS', MM)$

(* **Summary:** If the connection-establishment timer goes off, drop the connection (possibly *RST*ing the other end). *)

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$\mathrm{tcp\_drop\_and\_close}\ h.arch(\uparrow ETIMEDOUT)sock(sock', (oflgs, odata)) \wedge$

(* Note it should be the case that the socket is in *SYN_SENT*, and so *outsegs* will be empty, but that is not definite. *)

(* write to stream if possible *)
if exists_quad_of *sock* **then**
  let (i_1, p_1, i_2, p_2) = quad_of *sock* **in**
  $\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
  $\text{write}(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
  $SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$
else
  $SS' = SS$

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**

$(h\ \langle\!|socks := socks \oplus [(sid, sock)]|\!\rangle, SS, MM) \xrightarrow{\tau} (h\ \langle\!|socks := socks \oplus [(sid, sock')]|\!\rangle, SS', MM)$

$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$sock' = \text{tcp\_close}\ h.arch\ sock \wedge$

if exists_quad_of *sock* **then**
destroy(quad_of *sock*)$SS\ SS'$
else
$SS' = SS$

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 11

Host LTS: UDP Input Processing

11.1 Input Processing (UDP only)

11.1.1 Summary

deliver_in_udp_1	**udp: network nonurgent**	Get UDP datagram from host's in-queue and deliver it to a matching socket
deliver_in_udp_2	**udp: network nonurgent**	Get UDP datagram from host's in-queue but generate ICMP, as no matching socket
deliver_in_udp_3	**udp: network nonurgent**	Get UDP datagram from host's in-queue and drop as from a martian address

11.1.2 Rules

deliver_in_udp_1 **udp: network nonurgent** **Get UDP datagram from host's in-queue and deliver it to a matching socket**

$$(h_0, SS, MM) \xrightarrow{\tau} (h_0 \,\langle\!|iq := iq';\ socks := socks \oplus [(sid, sock\ \ pr := \text{UDP\_Sock}(rcvq'))]|\!\rangle,\ SS, MM)$$

$h_0 = h \,\langle\!|\ iq := iq;$
$\quad socks := socks \oplus$
$\quad\quad [(sid, sock\ \ pr := \text{UDP\_Sock}(rcvq))]|\!\rangle \wedge$
$rcvq' = rcvq \mathbin{@} [Dgram\_msg(\langle\!|\ data := data; is := \uparrow i_3; ps := ps_3|\!\rangle)] \wedge$
$dequeue\_iq(iq, iq', \uparrow(UDP(\langle\!|\ is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3; ps_2 := ps_4; data := data|\!\rangle))) \wedge$
$(\exists(ifid, ifd) :: (h_0.ifds).i_4 \ \in\ ifd.ipset) \wedge$
$sid \ \in\ \text{lookup\_udp}\ \ h_0.socks(i_3, ps_3, i_4, ps_4) h_0.bound\ h_0.arch \wedge$
$\mathbf{T}\wedge$ (* placeholder for "not a link-layer multicast or broadcast" *)
$\neg(is\_broadormulticast\ h_0.ifds\ i_4)\wedge$ (* seems unlikely, since $i_1 \ \in\ local\_ips\ h.ifds$ *)
$\neg(is\_broadormulticast\ h_0.ifds\ i_3)$

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 (p38) 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**

$(h\ \ iq := iq, SS, MM) \xrightarrow{\tau} (h\ \langle\!|iq := iq'; oq := \textbf{if}\ \ icmp\_to\_go\ \ \textbf{then}\ \ oq'\ \ \textbf{else}\ \ h.oq|\!\rangle, SS, MM)$

$dequeue\_iq(iq, iq', \uparrow(UDP(\langle\!| is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3;$
$ps_2 := ps_4; data := data|\!\rangle))) \wedge$
lookup_udp $h.socks(i_3, ps_3, i_4, ps_4)h.bound\ h.arch = \emptyset \wedge$
$icmp = ICMP(\langle\!| is_1 := \uparrow i_4; is_2 := \uparrow i_3; is3 := \uparrow i_3; is4 := \uparrow i_4;$
$ps_3 := ps_3; ps_4 := ps_4; proto := PROTO\_UDP; seq := *;$
$t := ICMP\_UNREACH(PORT)|\!\rangle) \wedge$
$(enqueue\_oq(h.oq, icmp, oq', \mathbf{T}) \vee icmp\_to\_go = \mathbf{F})$ (* non-deterministic ICMP generation *) $\wedge$
$i_4\ \in\ local\_ips\ h.ifds \wedge$
$\mathbf{T}\wedge$ (* placeholder for "not a link-layer multicast or broadcast" *)
$\neg(is\_broadormulticast\ h.ifds\ i_4)\wedge$ (* seems unlikely, since $i_1\ \in\ local\_ips\ h.ifds$ *)
$\neg(is\_broadormulticast\ h.ifds\ i_3)$

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 (p38) 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**

$(h\ \langle\!|iq := iq|\!\rangle, SS, MM) \xrightarrow{\tau} (h\ \langle\!|iq := iq'|\!\rangle, SS, MM)$

$dequeue\_iq(iq, iq', \uparrow(UDP\ \ dgram)) \wedge$
$dgram.is_2 = \uparrow\ i_2 \wedge$
$is_1 = dgram.is_1 \wedge$
$i_2\ \in\ local\_ips(h.ifds) \wedge$
$(\mathbf{F} \vee$
$\neg(\mathbf{T} \wedge$
$\neg(is\_broadormulticast\ h.ifds\ i_2)\wedge$ (* seems unlikely, since $i_1\ \in\ local\_ips\ h.ifds$ *)
$\neg(is_1 = *) \wedge$
$\neg is\_broadormulticast\ h.ifds(\textbf{the}\ \ is_1)$
)
)

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 12

Host LTS: ICMP Input Processing

12.1 Input Processing (ICMP only)

12.1.1 Summary

deliver_in_icmp_1	**all: network nonurgent**	Receive *ICMP_UNREACH_NET* etc for known socket
deliver_in_icmp_2	**all: network nonurgent**	Receive *ICMP_UNREACH_NEEDFRAG* for known socket
deliver_in_icmp_3	**all: network nonurgent**	Receive *ICMP_UNREACH_PORT* etc for known socket
deliver_in_icmp_4	**all: network nonurgent**	Receive *ICMP_PARAMPROB* etc for known socket
deliver_in_icmp_5	**all: network nonurgent**	Receive *ICMP_SOURCE_QUENCH* for known socket
deliver_in_icmp_6	**all: network nonurgent**	Receive and ignore other ICMP
deliver_in_icmp_7	**all: network nonurgent**	Receive and ignore invalid or unmatched ICMP

12.1.2 Rules

deliver_in_icmp_1 **all: network nonurgent** **Receive** *ICMP_UNREACH_NET* **etc for known socket**

$$(h_0, SS, MM) \xrightarrow{\tau} (h\ \langle\!|socks := socks \oplus [(sid, sock')]; iq := iq'; oq := oq'|\!\rangle, SS', MM)$$

$h_0 = h\ \langle\!|\ socks := socks \oplus [(sid, sock)];\ iq := iq;\ oq := oq|\!\rangle\ \wedge$
$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp))\ \wedge$
$icmp.t\ \in \{ICMP\_UNREACH\ c\ |$
$\quad c\ \in \{NET; HOST; SRCFAIL; NET\_UNKNOWN; HOST\_UNKNOWN; ISOLATED;$
$\quad\quad TOSNET; TOSHOST; PREC\_VIOLATION; PREC\_CUTOFF\}\}\ \wedge$
$icmp.is3 = \uparrow i_3\ \wedge$
$i_3\ \notin\ IN\_MULTICAST\ \wedge$
$sid\ \in\ \text{lookup\_icmp}\ h_0.socks\ icmp\ h_0.arch\ h_0.bound\ \wedge$

(**case** *sock.pr* **of**
  TCP_PROTO(*tcp_sock*) $\rightarrow$
    ($\exists$*icmp_seq.icmp.seq* = $\uparrow$ *icmp_seq* $\wedge$
    $\exists$*snd_una_le_icmp_seq* :: {**T**; **F**}.
    $\exists$*icmp_seq_lt_snd_max* :: {**T**; **F**}.
    $\exists$*cond* :: {**T**; **F**}.
    (*tcp_sock.cb.t_softerror* = $*$ $\implies$ *cond* = **F**) $\wedge$
    if *snd_una_le_icmp_seq* $\wedge$ *icmp_seq_lt_snd_max* **then**
      if *tcp_sock.st* = *ESTABLISHED* **then**
        sock′ = *sock*$\wedge$ (* ignore transient error while connected *)
        oq′ = *oq* $\wedge$
        SS′ = *SS*
      else if *tcp_sock.st* $\in$ {*CLOSED*; *LISTEN*; *SYN_SENT*; *SYN_RECEIVED*} $\wedge$
          cond **then**
        $\exists$*oflgs odata.* tcp_drop_and_close *h.arch*($\uparrow$ *EHOSTUNREACH*)*sock*(*sock*′, (*oflgs*, *odata*)) $\wedge$
        if exists_quad_of *sock* **then**
          let (i_1, p_1, i_2, p_2) = quad_of *sock* **in**
          $\exists S_0\ s\ s'.S_0 = S_0 \oplus$ [(streamid_of_quad$(i_1, p_1, i_2, p_2), s)$] $\wedge$
          write(i_1, p_1, i_2, p_2)(*oflgs*, *odata*)$s\ s'$ $\wedge$
          if *tcp_sock.st* = *CLOSED* **then**
            $SS' = S_0 \oplus$ [(streamid_of_quad$(i_1, p_1, i_2, p_2), s'$)]
          else
            destroy$(i_1, p_1, i_2, p_2)(S_0 \oplus$ [(streamid_of_quad$(i_1, p_1, i_2, p_2), s'$)])SS'
        else
          SS′ = *SS*
      else
        sock′ = *sock* ⦇ *pr* := TCP_PROTO(*tcp_sock*
                ⦇ *cb* := *tcp_sock.cb*
                  ⦇ *t_softerror* := $\uparrow$ *EHOSTUNREACH*⦈⦈)⦈ $\wedge$
        oq′ = *oq* $\wedge$
        SS′ = *SS*
    else
      (* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
      sock′ = *sock* $\wedge$
      oq′ = *oq* $\wedge$
      SS′ = *SS*) ‖
  UDP_PROTO(*udp_sock*) $\rightarrow$
    if *windows_arch h.arch* **then**
      sock′ = *sock* ⦇ *pr* := UDP_PROTO(*udp_sock*
            ⦇ *rcvq* := *udp_sock.rcvq* @ [(*Dgram_error*(⦇ *e* := *ECONNRESET*⦈))]⦈)⦈ $\wedge$ *oq*′ = *oq*
    else
      sock′ = *sock* ⦇ *es* :≙ $\uparrow$ *ECONNREFUSED*
            onlywhen((*sock.is*$_2 \neq *$) $\vee$ $\neg$(*SO_BSDCOMPAT* $\in$ *sock.sf.b*))⦈ $\wedge$ *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**

(h_0, SS, MM) $\xrightarrow{\tau}$ (*h* ⦇*socks* := *socks* $\oplus$
    [(*sid*, *sock*′)];
  iq := *iq*′;
  oq := *oq*′⦈,
  SS′, *MM*)

$h_0 = h$ ⦇ *socks* := *socks* $\oplus$

$[(sid, sock)];$
$iq := iq;$
$oq := oq\rrbracket \wedge$
$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp)) \wedge$
$icmp.t = ICMP\_UNREACH(NEEDFRAG\ icmpmtu) \wedge$
$(icmp.is3 = * \vee \mathbf{the}\ icmp.is3 \notin IN\_MULTICAST) \wedge$
$sid \in \text{lookup\_icmp}\ h_0.socks\ icmp\ h_0.arch\ h_0.bound \wedge$
let $nextmtu =$ **if F**$\wedge$ (* 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**
$\uparrow mtu \rightarrow \mathbf{w2n}\ mtu$
$\| * \rightarrow \text{next\_smaller}(\text{mtu\_tab}\ h_0.arch)curmtu$ **in**
if $nextmtu < 296$ **then**
(* Note this should lock curmtu in rmxcache; and not change rmxcache MTU from curmtu *)
$\uparrow curmtu$
else
(* Note here, *nextmtu* should be stored in rmxcache *)
$\uparrow nextmtu$
else
$*$ **in**
(**case** $sock.pr$ **of**
$\text{TCP\_PROTO}(tcp\_sock) \rightarrow$
$(\exists icmp\_seq.icmp.seq = \uparrow icmp\_seq \wedge$
if is_some $icmp.is3$ **then**
$\exists cond :: \{\mathbf{T}; \mathbf{F}\}.$
(**if** $cond$ **then**
if $nextmtu = *$ **then**
$sock' = sock \wedge$
$oq' = oq \wedge$
$SS' = SS$
else
$\exists tf\_doing\_tstmp :: \{\mathbf{T}; \mathbf{F}\}.$
let $mss = \mathbf{min}(sock.sf.n(SO\_SNDBUF))$
$(rounddown\ MCLBYTES$
$(\mathbf{the}\ nextmtu - 40 - (\mathbf{if}\ tf\_doing\_tstmp\ \mathbf{then}\ 12\ \mathbf{else}\ 0)))$ **in**
(* BSD: TS, plus NOOP for alignment *)
$\exists cond' :: \{\mathbf{T}; \mathbf{F}\}.$
if $cond'$ **then**
let $sock'' = sock$ **in**
$\exists sock'''\ FINs\ tcp\_sock'''.$
$sock'''.pr = \text{TCP\_PROTO}(tcp\_sock''') \wedge$
$stream\_tcp\_output\_perhaps\ sock''(sock''', FINs) \wedge$
$stream\_enqueue\_or\_fail\_sock(tcp\_sock'''.st \notin \{CLOSED; LISTEN; SYN\_SENT\})$
$h.arch\ h.rttab\ h.ifds(sock.is_1, sock.is_2)$
$sock''\ sock'''\ sock' \wedge$
case $FINs$ **of** $* \rightarrow SS' = SS$
$\| \uparrow FIN \rightarrow$
let $oflgs = \llparenthesis SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := FIN; RST := \mathbf{F}\rrparenthesis$ **in**
let $(i_1, p_1, i_2, p_2) = \text{quad\_of}\ sock$ **in**
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
$\text{write}(i_1, p_1, i_2, p_2)(oflgs, [\,])s\ s' \wedge$
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$
else
$sock' = sock \wedge oq' = oq \wedge SS' = SS$
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)

$sock' = sock \wedge oq' = oq \wedge SS' = SS)$
else
$sock' = sock \wedge oq' = oq \wedge SS' = SS) \parallel$
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*⦈))]⦈)⦈ $\wedge\ oq' = oq$
else
$sock' = sock$ ⦇ *es* := ↑ *EMSGSIZE*⦈ $\wedge\ oq' = oq)$

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_MSGSIZE.

---

deliver_in_icmp_3 <u>**all: network nonurgent**</u> **Receive** *ICMP_UNREACH_PORT* **etc for known socket**

(h_0, SS, MM) $\xrightarrow{\tau}$ (h ⦇*socks* := *socks* ⊕
[(*sid*, *sock'*)];
iq := *iq'*;
oq := *oq'*⦈,
SS', *MM*)

$h_0 = h$ ⦇ *socks* := *socks* ⊕
[(*sid*, *sock*)];
iq := *iq*;
oq := *oq*⦈ ∧
dequeue_iq(*iq*, *iq'*, ↑(*ICMP icmp*)) ∧
icmp.t ∈ {*ICMP_UNREACH c* |
c ∈ {*PROTOCOL*; *PORT*; *NET_PROHIB*; *HOST_PROHIB*; *FILTER_PROHIB*}} ∧
icmp.is3 = ↑ i_3 ∧
i_3 ∉ *IN_MULTICAST* ∧
sid ∈ lookup_icmp h_0.*socks icmp* h_0.*arch* h_0.*bound* ∧
(**case** *sock.pr* **of**
TCP_PROTO(*tcp_sock*) →
(∃*icmp_seq.icmp.seq* = ↑ *icmp_seq* ∧
∃*cond* :: {**T**; **F**}.
if *cond* **then**
if *tcp_sock.st* = *SYN_SENT* **then**
∃*oflgsodata*.
(* know from definition of tcp_drop_and_close that no segs will be emitted *)
tcp_drop_and_close *h.arch*(↑ *ECONNREFUSED*)*sock*(*sock'*, *oflgsodata*) ∧
null_flgs_data *oflgsodata* ∧
if exists_quad_of *sock* **then**
destroy(quad_of *sock*)*SS SS'*
else
SS' = *SS*
else
$sock' = sock \wedge oq' = oq \wedge SS' = SS$
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
$sock' = sock \wedge oq' = oq \wedge SS' = SS) \parallel$
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
$sock' = sock$ ⦃ $es :\hat{=} \uparrow(ECONNREFUSED)$
onlywhen$((sock.is_2 \neq *) \vee \neg(SO\_BSDCOMPAT \in sock.sf.b))$⦄ $\wedge\ oq' = oq))$

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**

$(h_0, SS, MM) \xrightarrow{\tau} (h$ ⦃$socks := socks \oplus$
$[(sid, sock')]$;
$iq := iq'$;
$oq := oq'$⦄,
$SS', MM)$

$h_0 = h$ ⦃ $socks := socks \oplus$
$[(sid, sock)]$;
$iq := iq$;
$oq := oq$⦄ $\wedge$
$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp)) \wedge$
$icmp.t \in \{ICMP\_PARAMPROB\ c \mid$
$c \in \{BADHDR; NEEDOPT\}\} \wedge$
$icmp.is3 = \uparrow i_3 \wedge$
$i_3 \notin IN\_MULTICAST \wedge$
$sid \in$ lookup_icmp $h_0.socks\ icmp\ h_0.arch\ h_0.bound \wedge$
(**case** $sock.pr$ **of**
TCP_PROTO$(tcp_sock) \rightarrow$
$(\exists icmp\_seq.icmp.seq = \uparrow icmp\_seq \wedge$
$\exists cond :: \{\mathbf{T}; \mathbf{F}\}$.
if $cond$ **then**
$\exists cond' :: \{\mathbf{T}; \mathbf{F}\}$.
$cond' \implies tcp\_sock.cb.t\_softerror \neq * \wedge$
if $tcp\_sock.st \in \{CLOSED; LISTEN; SYN\_SENT; SYN\_RECEIVED\} \wedge$
$cond'$ **then**
$\exists oflgs\ odata$.
tcp_drop_and_close $h.arch(\uparrow ENOPROTOOPT)sock(sock', (oflgs, odata)) \wedge$
if exists_quad_of $sock$ **then**
let $(i_1, p_1, i_2, p_2) =$ quad_of $sock$ **in**
$\exists S_0\ s\ s'.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
write$(i_1, p_1, i_2, p_2)(oflgs, odata)s\ s' \wedge$
if $tcp\_sock.st = CLOSED$ **then**
$SS' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')]$
else
destroy$(i_1, p_1, i_2, p_2)(S_0 \oplus [(\text{streamid_of_quad}(i_1, p_1, i_2, p_2), s')])SS'$
else
$SS' = SS$
else

$sock' = sock$ ⦃ $pr :=$ TCP_PROTO$(tcp_sock$
⦃ $cb := tcp\_sock.cb$ ⦃ $t\_softerror := \uparrow ENOPROTOOPT$⦄⦄$)$⦄ $\wedge$
$oq' = oq \wedge$
$SS' = SS$
else

$sock' = sock \wedge oq' = oq \wedge SS' = SS) \parallel$

$\mathrm{UDP\_PROTO}(udp\_sock) \rightarrow$
$(\textbf{if}\ \ windows\_arch\ h.arch\ \ \textbf{then}$
$sock' = sock\ \langle\!|\ pr := \mathrm{UDP\_PROTO}(udp\_sock$
$\langle\!|\ rcvq := udp\_sock.rcvq\ @\ [(Dgram\_error(\langle\!|\ e := ENOPROTOOPT|\!\rangle))]|\!\rangle)|\!\rangle \wedge$
$oq' = oq$
else
$sock' = sock\ \langle\!|\ es := \uparrow(ENOPROTOOPT)|\!\rangle \wedge oq' = oq))$

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_PARAMPROB.

deliver_in_icmp_5 **all: network nonurgent** **Receive** *ICMP_SOURCE_QUENCH* **for known socket**

$(h_0, SS, MM) \xrightarrow{\tau} (h\ \langle\!|socks := socks \oplus$
$[(sid, sock')];$
$iq := iq'|\!\rangle,$
$SS, MM)$

$h_0 = h\ \langle\!|\ socks := socks \oplus$
$[(sid, sock)];$
$iq := iq|\!\rangle \wedge$
$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp)) \wedge$
$icmp.t = ICMP\_SOURCE\_QUENCH\ \ QUENCH \wedge$
$icmp.is3 = \uparrow i_3 \wedge$
$i_3 \notin IN\_MULTICAST \wedge$
$sid \in \mathrm{lookup\_icmp}\ \ h_0.socks\ icmp\ h_0.arch\ h_0.bound \wedge$
$(\textbf{case}\ \ sock.pr\ \ \textbf{of}$
$\mathrm{TCP\_PROTO}(tcp\_sock) \rightarrow$
$(\exists icmp\_seq.icmp.seq = \uparrow icmp\_seq \wedge$
$\exists cond :: \{\mathbf{T}; \mathbf{F}\}.$
$\textbf{if}\ \ cond\ \ \textbf{then}$
$sock' = sock$
(* 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 *)
$sock' = sock) \ \|$
$\mathrm{UDP\_PROTO}(udp\_sock) \rightarrow$
$(\textbf{if}\ \ windows\_arch\ h.arch\ \ \textbf{then}$
$sock' = sock\ \langle\!|\ pr := \mathrm{UDP\_PROTO}(udp\_sock$
$\langle\!|\ rcvq := udp\_sock.rcvq\ @\ [(Dgram\_error(\langle\!|\ e := EHOSTUNREACH|\!\rangle))]|\!\rangle)|\!\rangle$
else
$sock' = sock\ \langle\!|\ es := \uparrow(EHOSTUNREACH)|\!\rangle))$

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_QUENCH.

deliver_in_icmp_6 **all: network nonurgent** **Receive and ignore other ICMP**

$(h\ \langle\!|iq := iq|\!\rangle, SS, MM) \xrightarrow{\tau} (h\ \langle\!|iq := iq'|\!\rangle, SS, MM)$

$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp)) \wedge$
$(icmp.t \in \{ICMP\_TIME\_EXCEEDED\ INTRANS; ICMP\_TIME\_EXCEEDED\ REASS\} \vee$
$icmp.t \in \{ICMP\_UNREACH(OTHER\ x) \mid x \in UNIV\} \vee$

$icmp.t \in \{ICMP\_SOURCE\_QUENCH(OTHER\ x) \mid x \in UNIV\} \lor$
$icmp.t \in \{ICMP\_TIME\_EXCEEDED(OTHER\ x) \mid x \in UNIV\} \lor$
$icmp.t \in \{ICMP\_PARAMPROB(OTHER\ x) \mid x \in UNIV\})$

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**

$(h\ \langle\!\langle iq := iq \rangle\!\rangle, SS, MM) \xrightarrow{\tau} (h\ \langle\!\langle iq := iq' \rangle\!\rangle, SS, MM)$

$dequeue\_iq(iq, iq', \uparrow(ICMP\ icmp)) \land$
$(icmp.t \in \{ICMP\_UNREACH\ c \mid \neg\exists x.c = OTHER\ x\} \lor$
$icmp.t \in \{ICMP\_PARAMPROB\ c \mid c \in \{BADHDR; NEEDOPT\}\} \lor$
$icmp.t = ICMP\_SOURCE\_QUENCH\ QUENCH) \land$
$(\textbf{if}\ \exists icmpmtu.icmp.t = ICMP\_UNREACH(NEEDFRAG\ icmpmtu)\ \textbf{then}$
$\quad \exists i_3.icmp.is3 = \uparrow i_3 \land i_3 \in IN\_MULTICAST$
$\textbf{else}$
$\quad (icmp.is3 = * \lor$
$\quad\quad \textbf{the}\ icmp.is3 \in IN\_MULTICAST \lor$
$\quad\quad \neg(\exists(sid, s) :: (h.socks).$
$\quad\quad\quad s.is_1 = icmp.is3 \land s.is_2 = icmp.is4 \land$
$\quad\quad\quad s.ps_1 = icmp.ps3 \land s.ps_2 = icmp.ps4 \land$
$\quad\quad\quad \text{proto\_of}\ s.pr = icmp.proto)))$

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 13

Host LTS: Network Input and Output

13.1 Input and Output (Network only)

13.1.1 Summary

deliver_in_99	**all: network nonurgent**	Really receive things
deliver_in_99a	**all: network nonurgent**	Ignore things not for us
deliver_out_99	**all: network nonurgent**	Really send things
deliver_loop_99	**all: network nonurgent**	Loop back a loopback message

13.1.2 Rules

deliver_in_99 **all: network nonurgent** **Really receive things**

$$(h\ \langle\!|iq := iq|\!\rangle, SS, MM) \xrightarrow{lbl} (h\ \langle\!|iq := iq'|\!\rangle, SS, MM')$$

$$\begin{array}{l}
(lbl = \tau\ \wedge \\
\quad MM' = MM\ \wedge \\
\quad (\exists q\ d\ q'\ d'\ tcp_segment. \\
\quad\quad iq = (q)_d\ \wedge \\
\quad\quad iq' = (q')_{d'}\ \wedge \\
\quad\quad enqueue_iq(iq, TCP\ tcp_segment, (q')_{d'}, queued))) \\
\vee \\
(lbl = msg\ \wedge \\
\quad MM = BAG_INSERT\ msg\ MM'\ \wedge \\
\quad sane_msg\ msg\ \wedge \\
\quad \uparrow i_1 = msg.is_2\ \wedge \\
\quad i_1\ \in\ local_ips(h.ifds)\ \wedge \\
\quad enqueue_iq(iq, msg, iq', queued))
\end{array}$$

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**

$(h\ \langle\!| iq := iq |\!\rangle, SS, BAG\_INSERT\ msg\ MM) \xrightarrow{msg} (h\ \langle\!| iq := iq' |\!\rangle, SS, BAG\_INSERT\ msg\ MM)$

$\uparrow i_1 = msg.is_2 \wedge$
$i_1 \notin local\_ips(h.ifds) \wedge$
$iq = iq'$

Description Do not accept messages that are not for this host.

deliver_out_99 **all: network nonurgent** **Really send things**

$(h\ \langle\!| oq := oq |\!\rangle, SS, MM) \xrightarrow{lbl} (h\ \langle\!| oq := oq' |\!\rangle, SS, MM')$

$(lbl = \tau \wedge$
$MM' = MM \wedge$

$(\exists q\ d\ tcp\_segment.$
$oq = (q)_d \wedge$
$dequeue\_oq((TCP\ tcp\_segment :: q)_d, oq', \uparrow(TCP\ tcp\_segment))))$
$\vee$
$(lbl = \overline{msg} \wedge$
$MM' = BAG\_INSERT\ msg\ MM \wedge$
$dequeue\_oq(oq, oq', \uparrow msg) \wedge$
$(\exists i_2.msg.is_2 = \uparrow i_2 \wedge i_2 \notin local\_ips\ h.ifds))$

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**

$(h\ \langle\!| iq := iq;$ $\xrightarrow{lbl}$ $(h\ \langle\!| iq := iq';$
$oq := oq |\!\rangle,$ $oq := oq' |\!\rangle,$
$SS, MM)$ $SS, MM)$

$(lbl = \tau \wedge$

$(\exists q\ d\ tcp\_segment.$
$oq = (q)_d \wedge$
$dequeue\_oq((TCP\ tcp\_segment :: q)_d, oq', \uparrow(TCP\ tcp\_segment))) \wedge$

$(\exists q\ d\ q'\ d'\ tcp\_segment.$
$iq = (q)_d \wedge$
$iq' = (q)_{d'} \wedge$
$enqueue\_iq(iq, TCP\ tcp\_segment, (q')_{d'}, queued)))$
$\vee$
$(dequeue\_oq(oq, oq', \uparrow msg) \wedge$
$(\exists i_2.msg.is_2 = \uparrow i_2 \wedge i_2 \in local\_ips\ h.ifds) \wedge$
$(lbl = \textbf{if}\ windows\_arch\ h.arch\ \textbf{then}\ \tau$

$\quad\quad$ **else** $\overleftrightarrow{msg}) \wedge$
$enqueue\_iq(iq, msg, iq', queued))$

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 14

Host LTS: BSD Trace Records and Interface State Changes

14.1 Trace Records and Interface State Changes (BSD only)

14.1.1 Summary

trace_1	**all: misc nonurgent**	Trace TCPCB state, *ESTABLISHED* or later
trace_2	**all: misc nonurgent**	Trace TCPCB state, pre-*ESTABLISHED*
interface_1	**all: misc nonurgent**	Change connectivity

14.1.2 Rules

trace_1 **all: misc nonurgent** **Trace TCPCB state, *ESTABLISHED* or later**

$$(h, SS, MM) \xrightarrow{\text{LH\_TRACE } tr} (h, SS, MM)$$

$sid \in \mathbf{dom}(h.socks) \wedge$
$tr = (flav, sid, quad, st, cb) \wedge$
$st \in \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING;$
$CLOSE\_WAIT; LAST\_ACK; TIME\_WAIT\} \wedge$
$\text{tracesock\_eq } tr\ sid(h.socks[sid])$

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 (p22) and tracecb_eq (p22) 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***

$$(h, SS, MM) \xrightarrow{\text{LH\_TRACE } tr} (h, SS, MM)$$

$sid \in \mathbf{dom}(h.socks) \wedge$
$tr = (flav, sid, quad, st, cb) \wedge$
$st \notin \{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING;$

$CLOSE\_WAIT; LAST\_ACK; TIME\_WAIT\} \wedge$
$(st = CLOSED \vee$ (* BSD emits one of these each time a tcpcb is created, eg at end of 3WHS *)
$((\exists sock\ tcp\_sock.$
$sock = (h.socks[sid]) \wedge$
$\text{proto\_of}\ sock.pr = PROTO\_TCP \wedge$
$tcp\_sock = \text{tcp\_sock\_of}\ sock \wedge$
$(\textbf{case}\ quad\ \textbf{of}$
$\uparrow(is_1, ps_1, is_2, ps_2) \rightarrow \textbf{if}\ flav = TA\_DROP \vee tcp\_sock.st = CLOSED\ \textbf{then}\ \mathbf{T}$
$\textbf{else}$
$is_1 = sock.is_1 \wedge ps_1 = sock.ps_1 \wedge is_2 = sock.is_2 \wedge ps_2 = sock.ps_2 \parallel$
$* \rightarrow \mathbf{T}) \wedge$
$(st = tcp\_sock.st \vee tcp\_sock.st = CLOSED))))$

---

interface_1 **all: misc nonurgent** **Change connectivity**

$$(h\ \langle\!| ifds := ifds |\!\rangle, SS, MM) \xrightarrow{\text{LH\_INTERFACE}(ifid, up)} (h\ \langle\!| ifds := ifds' |\!\rangle, SS, MM)$$

$ifid \in \mathbf{dom}(ifds) \wedge$
$ifds' = ifds \oplus (ifid, (ifds[ifid])\langle\!| up := up |\!\rangle)$

Description Allow interfaces to be externally brought up or taken down.

Chapter 15

Host LTS: Time Passage

15.1 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 network transition rules below.

15.1.1 Summary

Time_Pass_timedoption	time passes for an *′a timed* option value
Time_Pass_tcpcb	time passes for a tcp control block
Time_Pass_socket	time passes for a socket
fmap_every	apply *f* to range of finite map, and succeed if each application succeeds
fmap_every_pred	apply *f* to range of finite map, and succeed if each application succeeds
Time_Pass_host	time passes for a host
sowriteable	check whether a socket is writable
soreadable	check whether a socket is readable

15.1.2 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 : $duration \rightarrow$ tcpcb $\rightarrow$ tcpcb set option)(* recall: 'a set == 'a -> bool *)
$dur\ cb$
$=$ **let** $tt\_keep' =$ Time_Pass_timedoption $dur\ cb.tt\_keep$
in
if **is_some** tt_keep'
then
$\uparrow(\lambda cb'.$
$cb' =$
cb ⦃ (* not going to list everything here; too much! *)
$tt\_keep :=$ **the** tt_keep'
⦄)
else
$*$

– **time passes for a socket :**
(Time_Pass_socket : $duration \rightarrow$ socket $\rightarrow$ socket set option)
$dur\ s$
$=$ **case** $s.pr$ **of** UDP_PROTO(udp) $\rightarrow \uparrow\{s\}$
‖ TCP_PROTO(tcp_s) $\rightarrow$
let $cb's =$ Time_Pass_tcpcb $dur\ tcp\_s.cb$
in
if **is_some** $cb's$
then
$\uparrow(\lambda s'.$
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 : $('a \rightarrow 'b$ option$) \rightarrow ('c \mapsto 'a) \rightarrow ('c \mapsto 'b)$ option)
$f\ fm =$
let $fm' = f\ o\_f\ fm$
in
if $* \in$ **rng**(fm')
then $*$
else $\uparrow$(**the** $o\_f\ fm'$)

– **apply f to range of finite map, and succeed if each application succeeds :**
(fmap_every_pred : $('a \rightarrow 'b$ set option$) \rightarrow ('c \mapsto 'a) \rightarrow ('c \mapsto 'b)$set option)
$f\ fm =$
if $\exists y.y \in$ **rng**$(fm) \wedge f\ y = *$ **then**
$*$
else
$\uparrow\{fm' \mid$ **dom**$(fm) =$ **dom**$(fm') \wedge$
$\forall x.x \in$ **dom**$(fm) \implies fm'[x] \in$ (**the**$(f(fm[x])))\}$

– **time passes for a host :**
(Time_Pass_host : $duration \rightarrow$ host $\rightarrow$ host set option)
$dur\ h$

$= \mathbf{let}\ \ ts' = \text{fmap\_every}(\mathit{Time\_Pass\_timed}\ dur)h.ts$
$\mathbf{and}\ \ socks's = \text{fmap\_every\_pred}(\text{Time\_Pass\_socket}\ \ dur)h.socks$
$\mathbf{and}\ \ iq' = \mathit{Time\_Pass\_timed}\ dur\ h.iq$
$\mathbf{and}\ \ oq' = \mathit{Time\_Pass\_timed}\ dur\ h.oq$
$\mathbf{and}\ \ ticks's = \mathit{Time\_Pass\_ticker}\ dur\ h.ticks$
in
if $\mathbf{is\_some}\ ts' \land$
$\mathbf{is\_some}\ socks's \land$
$\mathbf{is\_some}\ iq' \land$
$\mathbf{is\_some}\ oq'$
then
$\uparrow(\lambda h'.$
$\mathbf{choose}\ socks' :: \mathbf{the}\ socks's.$
$\mathbf{choose}\ ticks' :: ticks's.$
$h' =$
h ⦃ (* arch unchanged *)
(* ifds unchanged *)
$ts := \mathbf{the}\ ts'$;
(* files unchanged *)
$socks := socks'$;
(* listen unchanged *)
(* bound unchanged *)
$iq := \mathbf{the}\ iq'$;
$oq := \mathbf{the}\ oq'$;
$ticks := ticks'$
(* fds unchanged *)
⦄)
else
$*$

– **check whether a socket is writable :**
sowriteable $arch\ sock\ SS\ b =$
case $sock.pr$ **of**
$\text{TCP\_PROTO}(tcp) \to ($
$\exists sndq.$
$(\mathbf{if}\ \ \text{exists\_quad\_of}\ \ sock\ \ \mathbf{then}$
$\mathbf{let}\ (i_1, p_1, i_2, p_2) = \text{quad\_of}\ \ sock\ \ \mathbf{in}$
$\exists S_0\ s.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \land$
$\exists peek\ inline\ flgs\ s'.$
$\text{read}(i_2, p_2, i_1, p_1)peek\ inline(flgs, sndq)s\ s'$
else
$sndq = [\,]) \land$
$b = ($
$((tcp.st\ \in \{\mathit{ESTABLISHED}; \mathit{CLOSE\_WAIT}\} \land$
$sock.sf.n(\mathit{SO\_SNDBUF}) - \mathbf{length}\ \ sndq \geq sock.sf.n(\mathit{SO\_SNDLOWAT})) \lor$ (* change to send_buffer_space *)
$(\mathbf{if}\ \ linux\_arch\ arch\ \ \mathbf{then}\ \neg sock.cantsndmore\ \ \mathbf{else}\ \ sock.cantsndmore) \lor$
$(linux\_arch\ arch \land tcp.st = \mathit{CLOSED}) \lor$
$sock.es \neq *))) \parallel$
$\text{UDP\_PROTO}(udp) \to \mathbf{T}$

Variations

Linux	On all OSes, attempting to write to a closed socket yields an immediate error. Only on Linux, however, does sowriteable return **T** in this case. On Linux, if the outgoing half of the connection has been closed by the application, the socket becomes non-writeable, whereas on other OSes it becomes writeable (because an immediate error would result from writing).

Rule version: $ Id: TCP3_hostLTSScript.sml,v 1.39 2009/02/20 13:08:08 tjr22 Exp $

– **check whether a socket is readable :**

soreadable *arch sock SS b* =
case *sock.pr* **of**
TCP_PROTO(*tcp*) → (
  $\exists rcvq.$
  (**if** exists_quad_of *sock* **then**
    let (i_1, p_1, i_2, p_2) = quad_of *sock* **in**
    $\exists S_0\ s.SS = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
    $\exists peek\ inline\ flgs\ s'.$
    $\text{read}(i_1, p_1, i_2, p_2) peek\ inline(flgs, rcvq) s\ s'$
  else
    $rcvq = [\,]) \wedge$
  $b = ($
    (**length** $rcvq \geq sock.sf.n(SO\_RCVLOWAT) \vee$
    $sock.cantrcvmore \vee$
    $(linux\_arch\ arch \wedge tcp.st = CLOSED) \vee$
    $(tcp.st = LISTEN \wedge$
      $\exists lis.tcp.lis = \uparrow lis \wedge$
        $lis.q \neq [\,]) \vee$
    $sock.es \neq *))) \parallel$
UDP_PROTO(*udp*) →
  $b = (udp.rcvq \neq [\,] \vee sock.es \neq * \vee (sock.cantrcvmore \wedge \neg windows\_arch\ arch))$

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 soreadable return **T** in this case.
WinXP	The socket will not be readable if it has been shutdown for reading.

Part VII

TCP3_stream

Chapter 16

Stream auxiliary functions

This file gives default initial values for stream types, and defines auxiliary functions, such as reading and writing to streams, and destroying one or more streams from a stream map.

16.1 Default initial values (TCP and UDP)

Default initial values for stream types.

16.1.1 Summary

initial_streamFlags	initial stream flags
initial_stream	initial unidirectional stream
initial_streams	initial bidirectional stream
streamid_of_quad	form the stream identifier from quad

16.1.2 Rules

– **initial stream flags :**

$$\text{initial\_streamFlags} = \langle\!|\ SYN := \mathbf{F};\ SYNACK := \mathbf{F};\ FIN := \mathbf{F};\ RST := \mathbf{F}\ |\!\rangle$$

Description

The initial flags are all false, since no messages are in transit.

– **initial unidirectional stream :**

$$\text{initial\_stream}(i, p)\, destroyed = \langle\!|\ i := i;\ p := p;\ flgs := \text{initial\_streamFlags};\ data := [\,];\ destroyed := destroyed\ |\!\rangle$$

Description

A unidirectional stream is constructed by giving the originating ip address and port, and the value the *destroyed* flag should take. Then *data* is initialized to the empty list.

– **initial bidirectional stream :**
initial_streams$(i_1, p_1, i_2, p_2) = ($
(* in stream is initially destroyed because other host knows nothing of the connection attempt *)
let *in_* = initial_stream(i_2, p_2)**T in**
let *out* = initial_stream(i_1, p_1)**F in**
$\langle\!|$ *streams* :=$\{$*in_*; *out*$\}|\!\rangle)$

Description

A stream is constructed based on the quad (i_1, p_1, i_2, p_2). Only one endpoint, at the originating host (i_1, p_1), exists, thus, the output stream is not destroyed, whilst the input stream is destroyed.

– **form the stream identifier from quad :**
streamid_of_quad$((i_1, p_1, i_2, p_2)$: *ip#port#ip#port*$) = \{(i_1, p_1); (i_2, p_2)\}$

Description

A stream identifier is an unordered pair of the endpoint ip and port addresses.

16.2 Auxiliary functions (TCP and UDP)

Auxiliary stream functions, such as reading and writing to a stream.

16.2.1 Summary

null_flgs_data	flags and data corresponding to no control information
make_syn_flgs_data	flags and data corresponding to an initial *SYN* message
make_syn_ack_flgs_data	flags and data corresponding to an initial *SYNACK* message
sync_streams	retrieve unidirectional streams from bidirectional stream
write	write flags and data to a stream
read	read flags and data from a stream

16.2.2 Rules

– **flags and data corresponding to no control information :**
null_flgs_data(*flgs*, *data*) = (
flgs =$\langle\!|$ *SYN* := **F**; *SYNACK* := **F**; *FIN* := **F**; *RST* := **F**$|\!\rangle$ ∧
data = [ ])

– **flags and data corresponding to an initial *SYN* message :**
make_syn_flgs_data(*flgs*, *data* : *char* list) = (
flgs =$\langle\!|$ *SYN* := **T**; *SYNACK* := **F**; *FIN* := **F**; *RST* := **F**$|\!\rangle$ ∧
data = [ ])

– **flags and data corresponding to an initial** $SYNACK$ **message :**
make_syn_ack_flgs_data($flgs$, $data$: $char$ list) = (
$flgs = \langle\!| SYN := \mathbf{F}; SYNACK := \mathbf{T}; FIN := \mathbf{F}; RST := \mathbf{F} |\!\rangle \wedge$
$data = [\,]$)

– **retrieve unidirectional streams from bidirectional stream :**
sync_streams($i_1 : ip, p_1 : port, i_2 : ip, p_2 : port$)($s$: tcpStreams)($in\_, out$) = (
$s.streams = \{in\_; out\} \wedge$
$(in\_.i, in\_.p) = (i_2, p_2) \wedge$
$(out.i, out.p) = (i_1, p_1)$)

(* i1 p1 are local, i2 p2 are foreign *)

Description

A function to extract the input stream $in\_$ and output stream out from a bidirectional stream s based on the ip address and port of an endpoint.

– **write flags and data to a stream :**
write(i_1, p_1, i_2, p_2)($flgs, data$)$s\ s'$ = (
$\exists in\_\ out\ in'\ out'$.
sync_streams(i_1, p_1, i_2, p_2)$s(in\_, out) \wedge$
sync_streams(i_1, p_1, i_2, p_2)$s'(in', out') \wedge$
$in' = in\_ \wedge$
$out'.flgs =$
$\langle\!| SYN := (out.flgs.SYN \vee flgs.SYN);$
$SYNACK := (out.flgs.SYNACK \vee flgs.SYNACK);$
$FIN := (out.flgs.FIN \vee flgs.FIN);$
$RST := (out.flgs.RST \vee flgs.RST)$
$|\!\rangle \wedge$
$out'.data = (out.data +\!+ data)$)

Description

The unidirectional streams before ($in\_, out$) and after (in', out') are first extracted using sync_streams. The $flgs$ and $data$ of the output stream out' are updated to reflect the write. For example, $data$ is appended to $out.data$ to form $out'.data$.

– **read flags and data from a stream :**
read(i_1, p_1, i_2, p_2)($peek$: bool)($inline$: bool)($flgs$: streamFlags, $data$: $char$ list)$s\ s'$ = (
$\exists in\_\ out\ in'\ out'$.
sync_streams(i_1, p_1, i_2, p_2)$s(in\_, out) \wedge$
sync_streams(i_1, p_1, i_2, p_2)$s'(in', out') \wedge$
$out' = out \wedge$

(**case** $flgs.SYN$ **of** $\mathbf{T} \rightarrow in'.flgs.SYN = \mathbf{F} \wedge in\_.flgs.SYN = \mathbf{T} \parallel \mathbf{F} \rightarrow in'.flgs.SYN = in\_.flgs.SYN) \wedge$
(**case** $flgs.SYNACK$ **of**
$\mathbf{T} \rightarrow in'.flgs.SYNACK = \mathbf{F} \wedge in\_.flgs.SYNACK = \mathbf{T}$
$\parallel \mathbf{F} \rightarrow in'.flgs.SYNACK = in\_.flgs.SYNACK) \wedge$
(**case** $flgs.FIN$ **of** $\mathbf{T} \rightarrow in'.flgs.FIN = \mathbf{F} \wedge in\_.flgs.FIN = \mathbf{T} \parallel \mathbf{F} \rightarrow in'.flgs.FIN = in\_.flgs.FIN) \wedge$

$(\mathbf{case}\ \mathit{flgs.RST}\ \mathbf{of}\ \mathbf{T} \rightarrow \mathit{in}'.\mathit{flgs.RST} = \mathbf{F} \wedge \mathit{in\_}.\mathit{flgs.RST} = \mathbf{T} \parallel \mathbf{F} \rightarrow \mathit{in}'.\mathit{flgs.RST} = \mathit{in\_}.\mathit{flgs.RST}) \wedge$

$(\exists \mathit{pre}\ \mathit{post}.$
$((\mathit{pre} + +\mathit{data} + +\mathit{post}) = \mathit{in\_}.\mathit{data}) \wedge$
$(\mathit{inline} \implies \mathit{pre} = [\,]) \wedge$
$\mathbf{if}\ \mathit{peek}\ \mathbf{then}$
$\mathit{in}'.\mathit{data} = \mathit{in\_}.\mathit{data}$
$\mathbf{else}$
$\mathit{in}'.\mathit{data} = (\mathit{pre} + +\mathit{post})))$

Description

The unidirectional streams before ($\mathit{in\_}$, out) and after (in', out') are first extracted using sync_streams. The *flgs* and *data* of the input stream in' are updated to reflect the read. For example, if $\mathit{flgs.SYN}$ is set, a SYN was read, which causes the SYN flag for input stream in' to be lowered; furthore, $\mathit{in\_}.\mathit{flgs.SYN}$ must also have been set, i.e. there must have been a SYN to read.

16.3 Stream removal (TCP and UDP)

Auxiliary functions to help with removing streams when they have been destroyed.

16.3.1 Summary

both_streams_destroyed	test whether both unidirectional streams are destroyed
remove_destroyed_streams	restrict the stream map to those streams that are not destroyed
destroy	destroy a particular unidirectional stream, then clean up
destroy_quads	destroy all quads in a stream map, then clean up

16.3.2 Rules

– **test whether both unidirectional streams are destroyed :**
both_streams_destroyed $\mathit{ss} = \forall s\ t.\mathit{ss.streams} = \{s; t\} \implies s.\mathit{destroyed} \wedge t.\mathit{destroyed}$

– **restrict the stream map to those streams that are not destroyed :**
remove_destroyed_streams$(\mathit{SS} : \mathit{streamid} \mapsto \mathsf{tcpStreams}) = ($
$\mathbf{let}\ \mathit{alive} = \{\mathit{stid} \mid \neg\,\text{both\_streams\_destroyed}(\mathit{SS}[\mathit{stid}])\}\ \mathbf{in}$
$\mathit{SS}|_{\mathit{alive}})$

Description

Streams where both unidirectional streams are destroyed are garbage collected.

– **destroy a particular unidirectional stream, then clean up :**
destroy$(i_1, p_1, i_2, p_2)\mathit{SS}\ S'' = ($
$\exists S_0\ s\ \mathit{in\_}\ \mathit{out}\ s'\ S'.$
$\mathit{SS} = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s)] \wedge$
sync_streams$(i_1, p_1, i_2, p_2)s(\mathit{in_}, \mathit{out}) \wedge$
$s' = \langle\!| \mathit{streams} := \{\mathit{in\_}; \mathit{out}\ \langle\!| \mathit{destroyed} := \mathbf{T} |\!\rangle\} |\!\rangle \wedge$

$S' = S_0 \oplus [(\text{streamid\_of\_quad}(i_1, p_1, i_2, p_2), s')] \wedge$
$S'' = \text{remove\_destroyed\_streams}\ \ S')$

Description

The particular stream s identified by quad (i_1, p_1, i_2, p_2) is extracted from the stream map SS. In turn, the input and output streams are extracted from s. The stream s is updated to mark the output stream *out* as destroyed, producing the updated stream map S'. Finally, streams with both endpoints destroyed are garbage collected using remove_destroyed_streams.

– **destroy all quads in a stream map, then clean up :**

destroy_quads $quads(SS : streamid \mapsto \mathsf{tcpStreams})S'' = ($
$\exists S'.\mathbf{dom}(S') = \mathbf{dom}(SS) \wedge$
$(\forall stid.stid\ \in (\mathbf{dom}(SS)) \implies$
$\exists in\_\ out\ in'\ out'.$
$(SS[stid]).streams = \{in\_; out\} \wedge$
$in' = in\_ \llparenthesis\ destroyed\ :\hat{=}\ \mathbf{T}\ \mathbf{onlywhen}((in\_.i, in\_.p, out.i, out.p) \in\ quads)\rrparenthesis \wedge$
$out' = out \llparenthesis\ destroyed\ :\hat{=}\ \mathbf{T}\ \mathbf{onlywhen}((out.i, out.p, in\_.i, in\_.p) \in\ quads)\rrparenthesis \wedge$
$(S'[stid]).streams = \{in'; out'\}) \wedge$
$S'' = \text{remove\_destroyed\_streams}\ \ S')$

Description

Similar to destroy, but allowing the destruction of multiple streams, for example, when a listening socket with pending connections is closed.

Part VIII

TCP3_net

Chapter 17

Network labelled transition system

This file defines the network model, using the host LTS defined previously.

17.1 Basic network types (TCP and UDP)

Basic network types, and transition labels.

17.1.1 Summary

type_abbrev_hosts	
type_abbrev_streams	
type_abbrev_msgs	
type_abbrev_net	
Lnet0	net transition labels
rn	net transition rule names

17.1.2 Rules

– :
type_abbrev *hosts* : *hostid* ↦ host

– :
type_abbrev *streams* : *streamid* ↦ tcpStreams

– :
type_abbrev *msgs* : *msg multiset*

– :
type_abbrev *net* : *hosts#streams#msgs*

– net transition labels :
Lnet0 =
(* library interface *)
  LN_CALL **of** *hostid#tid#LIB_interface*
| LN_RETURN **of** *hostid#tid#TLang*

(* connectivity changes *)

| LN_INTERFACE **of** *hostid*#*ifid*#bool

(* miscellaneous *)
| LN_TAU
| LN_EPSILON **of** *duration*

– **net transition rule names :**
rn = *call* | *return* | *tau* | *interface* | *host_tau* | *time_pass* | *trace*

17.2 Network labelled transition system (TCP and UDP)

17.2.1 Summary

call
return
tau
interface
host_tau
time_pass
trace

17.2.2 Rules

call

$$((hs \oplus (hid, h), S, M) : net) \xrightarrow{(\textsc{Ln\_call}(hid, tid, c))} (hs \oplus (hid, h'), S', M')$$

$$(rn / * rp, rc * / (h, S, M) \xrightarrow{tid \cdot c} (h', S', M'))$$

Description
A thread *tid* on host *h* executes a sockets call *c* which does not sync with the streams.

return

$$((hs \oplus (hid, h), S, M) : net) \xrightarrow{(\textsc{Ln\_return}(hid, tid, v))} (hs \oplus (hid, h'), S', M')$$

$$(rn / * rp, rc * / (h, S, M) \xrightarrow{\overline{tid \cdot v}} (h', S', M'))$$

Description
A thread *tid* on host *h* returns from a sockets call.

tau

$$((hs, S, M) : net) \xrightarrow{(\textsc{Ln\_tau})} (hs, S, M)$$

T

Rule version:

Description

This tau action at the network level corresponds to the hosts doing a $\overline{msg}$ or a *msg* transition.

interface

$$((hs \oplus (hid, h), S, M) : net) \xrightarrow{(\text{Ln\_INTERFACE}(hid, ifid, up))} (hs \oplus (hid, h'), S', M')$$

$$(rn/*rp, rc*/(h, S, M) \xrightarrow{\text{Lh\_INTERFACE}(ifid, up)} (h', S', M'))$$

Description

Network interface change.

host_tau

$$((hs \oplus (hid, h), S, M) : net) \xrightarrow{\text{Ln\_TAU}} (hs \oplus (hid, h'), S', M')$$

$$(rn/*rp, rc*/(h, S, M) \xrightarrow{\tau} (h', S', M'))$$

Description

Allow a host to do a τ transition.

time_pass

$$((hs, S, M) : net) \xrightarrow{(\text{Ln\_EPSILON}\ dur)} (hs'', S, M)$$

$$(\forall h.h \in \mathbf{rng}(hs) \implies \neg(\exists rn\ rp\ rc\ lbl\ h'\ S'\ M'.$$
$$(rn/*rp, rc*/(h, S, M) \xrightarrow{lbl} (h', S', M')) \wedge is\_urgent\ rc)) \wedge$$

(* Time passes for the hosts. *)
$hs' = (\text{Time\_Pass\_host}\ \ dur)o\_f\ hs \wedge$
$\neg(* \in \mathbf{rng}(hs')) \wedge$

$\mathbf{dom}(hs'') = \mathbf{dom}(hs) \wedge$
$(\forall hid.hid \in \mathbf{dom}(hs) \implies hs''[hid] \in (\mathbf{the}(hs'[hid])))$

Description

Allow time to pass for hosts. The check $\neg(* \in \mathbf{rng}(hs'))$ ensures that time actually can pass for a host, i.e. that there are no urgent events that need to happen.

trace

$$((hs \oplus (hid, h), S, M) : net) \xrightarrow{(\text{Ln\_TAU})} (hs \oplus (hid, h'), S', M')$$

$hid \notin \mathbf{dom}(hs) \wedge$

$$(rn/*rp, rc*/(h, S, M) \xrightarrow{\text{LH\_TRACE } tr} (h', S', M'))$$

Description

Trace records give LN_TAU transitions at the network level.

Part IX

TCP3_absFun

Chapter 18

Abstraction function

This file defines the abstraction function, from protocol-level network states (and transition labels) to service-level network states (and transition labels).

18.1 Auxiliary functions (TCP and UDP)

Basic abstraction functions for basic TCP host types.

18.1.1 Summary

tcpcb1_to_3	abstract a tcpcb
tcp_socket1_to_3	abstract a tcp_socket
socket1_to_3	abstraction a socket
host1_to_3	abstract a host

18.1.2 Rules

– **abstract a** tcpcb **:**
(tcpcb1_to_3 : *TCP1_hostTypes* \$*tcpcb* → *TCP3_hostTypes* \$*tcpcb*)*cb* = (
⟪ *tt_keep* := *cb.tt_keep*;
t_softerror := *cb.t_softerror*
⟫)

– **abstract a** tcp_socket **:**
(tcp_socket1_to_3 : *TCP1_hostTypes* \$*tcp_socket* → *TCP3_hostTypes* \$*tcp_socket*)*s* = (
⟪ *st* := *s.st*;
cb := tcpcb1_to_3 *s.cb*;
lis := *s.lis*
⟫)

– **abstraction a** socket **:**
(socket1_to_3 : *TCP1_hostTypes* \$*socket* → *TCP3_hostTypes* \$*socket*)*s* = (
⟪ *fid* := *s.fid*;
sf := *s.sf*;
is_1 := $s.is_1$;
ps_1 := $s.ps_1$;
is_2 := $s.is_2$;
ps_2 := $s.ps_2$;
es := *s.es*;
cantsndmore := *s.cantsndmore*;

$cantrcvmore := s.cantrcvmore;$
$pr := (\mathbf{case}\ s.pr\ \mathbf{of}\ \mathrm{TCP\_PROTO}\ tcp\_sock \to \mathrm{TCP\_PROTO}(\mathrm{tcp\_socket1\_to\_3}\ tcp\_sock)$
$\| \mathrm{UDP\_PROTO}\ udp\_sock \to \mathrm{UDP\_PROTO}\ udp\_sock)$
$|\!\!\}))$

– **abstract a host :**
$(\mathrm{host1\_to\_3} : TCP1\_hostTypes\ \$host \to TCP3\_hostTypes\ \$host)h = ($
$\mathbf{let}\ filter\_non\_TCP\_msgs =$
$\lambda q.\mathbf{case}\ q\ \mathbf{of}\ (msgs)_d \to (\mathbf{filter}(\lambda msg.\mathbf{case}\ msg\ \mathbf{of}\ TCP\ \_1 \to \mathbf{F} \| \_2 \to \mathbf{T})msgs)_d$
$\mathbf{in}$
$\{\!|\ arch := h.arch;$
$privs := h.privs;$
$ifds := h.ifds;$
$rttab := h.rttab;$
$ts := h.ts;$
$files := h.files;$
$socks := \mathrm{socket1\_to\_3}\ o\_f\ h.socks;$
$listen := h.listen;$
$bound := h.bound;$
$iq := filter\_non\_TCP\_msgs\ h.iq;$
$oq := filter\_non\_TCP\_msgs\ h.oq;$
$bndlm := h.bndlm;$
$ticks := h.ticks;$
$fds := h.fds;$
$params := h.params$
$|\!\!\}))$

18.2 Stream reassembly (TCP and UDP)

For the case where the sender is absent, we have to recover the stream contents from segments on the wire, using a stream reassembly function.

18.2.1 Summary

stream_reass	reassemble the stream from segments on the wire

18.2.2 Rules

– **reassemble the stream from segments on the wire :**
$\mathrm{stream\_reass}(seq : tcpLocal\ seq32)(segs : tcpSegment\ \mathsf{set}) = ($
(* REMARK first arg should be word32 *)
$\mathbf{let}\ myrel = \{(i, c) \mid$
$\exists seg.seg \in segs \wedge$
$\mathbf{num}(i - seg.seq) < \mathbf{length}\ seg.data \wedge$
$c = EL(\mathbf{num}(i - seg.seq))seg.data\}\ \mathbf{in}$
$\mathbf{let}\ cs = \{(cs : \mathsf{byte\ list}) \mid$
$(\forall n : num.n < \mathbf{length}\ cs \implies myrel(seq + n, EL\ n\ cs)) \wedge$
$(\neg\exists c.(seq + (\mathbf{length}\ cs), c) \in myrel)\}\ \mathbf{in}$
$CHOICE\ cs)$

Description
This stream reassembly function is closely based on that defined in the protocol-level specification.

Rule version:

18.3 Abstraction function (TCP and UDP)

The full abstraction function builds on a unidirectional version. Both are presented in this section.

18.3.1 Summary

ERROR	a simple way to indicate that an error has occurred
abs_hosts_one_sided	unidirectional abstraction function
abs_hosts	the full abstraction function for host states
abs_lbl	abstract transition labels
abs_trans	abstract a full protocol-level network transition

18.3.2 Rules

– **a simple way to indicate that an error has occurred :**
$ERROR(a : {}'a) = (ARB : {}'b)$

– **unidirectional abstraction function :**
abs_hosts_one_sided$(i_1, p_1, i_2, p_2)(h, msgs, i) = ($

```
  (* get the messages that we are interested in, including those in oq and iq *)
 let (hoq, iiq) =
      case (h.oq, i.iq) of ((omsgs)_1 , (imsgs)_2) → (omsgs, imsgs) in
 let msgs = list_to_set hoq ∪ msgs ∪ (list_to_set iiq) in
  (* only consider TCP messages ... *)
 let msgs = {msg | TCP msg ∈ msgs} in
  (* ... that match the quad *)
 let msgs = msgs ∩
      {msg | msg = msg ⦇ is1 :=↑ i1; ps1 :=↑ p1; is2 :=↑ i2; ps2 :=↑ p2⦈} in

  (* pick out the send and receive sockets *)
 let smatch i1 p1 i2 p2 s = ((s.is1, s.ps1, s.is2, s.ps2) = (↑ i1, ↑ p1, ↑ i2, ↑ p2)) in
 let snd_sock = Punique_range(smatch i1 p1 i2 p2)h.socks in
 let rcv_sock = Punique_range(smatch i2 p2 i1 p1)i.socks in

 let tcpsock_of sock = case sock.pr of
      TCP1_hostTypes $TCP_PROTO tcpsock → tcpsock
    || _3 → ERROR“abs_hosts_one_sided:tcpsock_of”
  in

  (* the difficult part of the abstraction function is to compute data *)
 let (data : byte list) = case (snd_sock, rcv_sock) of
      (↑(_8, hsock), ↑(_9, isock)) → (
           let htcpsock = tcpsock_of hsock in
           let itcpsock = tcpsock_of isock in
           let (snd_una, sndq) = (htcpsock.cb.snd_una, htcpsock.sndq) in
           let (rcv_nxt, rcvq) = (itcpsock.cb.rcv_nxt, itcpsock.rcvq) in
           let rcv_nxt = tcp_seq_flip_sense rcv_nxt in
           let sndq′ = DROP((num(rcv_nxt − snd_una)))sndq in
           rcvq + +sndq′)

    || (↑(_8, hsock), ∗) → (
           let htcpsock = tcpsock_of hsock in
           htcpsock.sndq)
```

$\| (*, \uparrow(\_9, isock)) \to ($
  let $itcpsock = tcpsock\_of\ isock$ **in**
  let $(rcv\_nxt : tcpLocal\ seq32, rcvq : \mathsf{byte\ list}) =$
    $(tcp\_seq\_flip\_sense(itcpsock.cb.rcv\_nxt), itcpsock.rcvq)$ **in**
  $rcvq ++(\text{stream\_reass}\ rcv\_nxt\ msgs))$

$\| (*, *) \to ERROR$“abs_hosts_one_sided:data”
in
$\langle\!| i := i_1;$
$p := p_1;$
$flgs :=$
$\langle\!| SYN :=(\exists msg.msg \in msgs \wedge msg = msg \langle\!| SYN := \mathbf{T}; ACK := \mathbf{F}|\!\rangle);$
$SYNACK :=(\exists msg.msg \in msgs \wedge msg = msg \langle\!| SYN := \mathbf{T}; ACK := \mathbf{T}|\!\rangle);$
$FIN :=(\exists msg.msg \in msgs \wedge msg = msg \langle\!| FIN := \mathbf{T}|\!\rangle);$
$RST :=(\exists msg.msg \in msgs \wedge msg = msg \langle\!| RST := \mathbf{T}|\!\rangle)$
$|\!\rangle;$
$data := data;$
$destroyed :=($**case** snd_sock **of**
$\uparrow(sid, hsock) \to ((tcpsock\_of\ hsock).st = CLOSED)$
$\| * \to \mathbf{T})$
$|\!\rangle)$

Description

The core of the abstraction function is to compute the *data* in the stream, given the connection endpoints and the segments on the wire.

Normally the sender and receiver endpoints are both active. In this case, the sender *sndq* and the receiver *rcvq* contain bytes corresponding to sequence number intervals. These intervals overlap, so to recover the data in the stream, we must drop some data from the *sndq*. We drop $rcv\_nxt - snd\_una$ bytes and then append the resulting $sndq'$ to *rcvq* to form the contents of the stream.

The other cases are handled in a similar way. If the receiver endpoint is absent, the data is just that data in the sender's *sndq*. If the sender endpoint is absent, the data is reassembled from segments on the wire, using stream_reass.

The *flgs* are calculated based on the flags set in segments on the wire. In fact, this should also take into account segment validity, but currently this is not handled correctly at the protocol-level, and we want to maintain the invariant that every protocol-level trace maps to a service-level trace.

The destroyed flag is true iff the socket is *CLOSED* or no longer exists.

– **the full abstraction function for host states :**
abs_hosts$(i_1, p_1, i_2, p_2)(h1, msgs, h2) = ($
let $n1 =$ host1_to_3 $h1$ **in**
let $n2 =$ host1_to_3 $h2$ **in**
let $(streams : \mathsf{tcpStreams\ option}) =$
  let $s12 =$ abs_hosts_one_sided$(i_1, p_1, i_2, p_2)(h1, msgs, h2)$ **in**
  let $s21 =$ abs_hosts_one_sided$(i_2, p_2, i_1, p_1)(h2, msgs, h1)$ **in**
  $($**case** $s12.destroyed \wedge s21.destroyed$ **of**
    $\mathbf{T} \to *$
    $\| \mathbf{F} \to \uparrow\langle\!| streams :=\{s12; s21\}|\!\rangle)$
in
$(n1, streams, n2))$

Description

The abstraction function maps protocol-level host states and segments on the wire to service-level host states and streams. It uses the unidirectional abstraction function abs_hosts_one_sided twice to form streams *s12* and *s21*. If these streams are both destroyed, then the resulting *streams* (an option) is $*$, otherwise it is a pair of the unidirectional streams.

– **abstract transition labels :**
abs_lbl *lbl* = (**case** *lbl* **of**
  Ln0_call(*hid*, *tid*, *lib*) → LN_CALL(*hid*, *tid*, *lib*)
 ∥ *Ln0_return*(*hid*, *tid*, *tlang*) → LN_RETURN(*hid*, *tid*, *tlang*)
 ∥ *Ln0_interface*(*hid*, *ifid*, *up*) → *ERROR*"absfn: Ln0_interface"
 ∥ *Ln0_tau* → LN_TAU
 ∥ *Ln0_epsilon dur* → LN_EPSILON *dur*
 ∥ *Ln0_trace tr* → LN_TAU)

Description

The abstraction function must also map protocol-level transition labels to service-level transition labels. This is a straightforward bijection. Interface changes are not currently handled at the service level.

– **abstract a full protocol-level network transition :**
abs_trans$(i_1, p_1, i_2, p_2)(h1, msgs, h2) lbl(h1', msgs', h2') = ($
  let n = abs_hosts$(i_1, p_1, i_2, p_2)(h1, msgs, h2)$ **in**
  let n' = abs_hosts$(i_1, p_1, i_2, p_2)(h1', msgs', h2')$ **in**
  let *nlbl* = abs_lbl *lbl* **in**
  $(n, nlbl, n'))$

Description

The abs_trans function ties together the previous host and label abstraction functions to produce a service-level transition from a protocol-level transition.

Rule version:

Bibliography

[1] R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*, pages 169–181, 2001.

[2] E. Biagioni. A structured TCP in Standard ML. In *Proc. SIGCOMM '94*, pages 36–45, 1994.

[3] J. Billington and B. Han. On defining the service provided by TCP. In *Proc. ACSC: 26th Australasian Computer Science Conference*, Adelaide, 2003.

[4] A. Biltcliffe, M. Dales, S. Jansen, T. Ridge, and P. Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *Proc. ICNP, The 14th IEEE International Conference on Network Protocols*, Nov. 2006.

[5] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM 2005 (Philadelphia)*, Aug. 2005.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Technical Report UCAM-CL-TR-624, Computer Laboratory, University of Cambridge, Mar. 2005. 88pp. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Technical Report UCAM-CL-TR-625, Computer Laboratory, University of Cambridge, Mar. 2005. 386pp. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[8] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL'06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 2006. ACM Press.

[9] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997. Full version of a paper in SIGCOMM '96.

[10] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proc. TACAS'03, LNCS 2619*, pages 113–127, 2003.

[11] M. Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In M. D. Atkinson and F. K. H. A. Dehne, editors, *CATS*, volume 41 of *CRPIT*, pages 21–30. Australian Computer Society, 2005.

[12] E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In *Proc. 7th SPIN Workshop, LNCS 1885*, pages 187–204, 2000.

[13] R. Hofmann and F. Lemmen. Specification-driven monitoring of TCP/IP. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 2000.

[14] The HOL 4 system, Kananaskis-3 release. `hol.sourceforge.net`.

[15] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGGCOMM '99*, pages 3–13, August 1999.

[16] P. Li. *Programmable Concurrency in a Pure and Lazy Language.* PhD thesis, University of Pennsylvania, August 2008.

[17] P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *Proc. PLDI*, pages 189–199, 2007.

[18] N. Lynch and F. Vaangdrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[19] S. L. Murphy and A. U. Shankar. A verified connection management protocol for the transport layer. In *Proc. SIGCOMM*, pages 110–125, 1987.

[20] S. L. Murphy and A. U. Shankar. Service specification and protocol construction for the transport layer. In *Proc. SIGCOMM*, pages 88–97, 1988.

[21] M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, Sept. 2002.

[22] J. Postel. *A Graph Model Analysis of Computer Communications Protocols.* University of California, Computer Science Department, PhD Thesis, 1974.

[23] T. Ridge. Verifying distributed systems: the operational approach. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 429–440. ACM, 2009.

[24] I. Schieferdecker. Abruptly-terminated connections in TCP – a verification example. In *Proc. COST 247 International Workshop on Applied Formal Methods In System Design*, June 1996.

[25] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proc. TACS 2001: Fourth International Symposium on Theoretical Aspects of Computer Software, Tohoku University, Sendai*, Oct. 2001.

[26] M. A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.

[27] M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE IX/PSTV XVI*, pages 129–144, 1996.

[28] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proc. ESOP, LNCS 2305*, pages 278–294, Apr. 2002.

Index

abs_hosts, 278
abs_hosts_one_sided, 9, 277
abs_lbl, 279
abs_trans, 279
accept_1, 60
accept_2, 61
accept_3, 61
accept_4, 62
accept_5, 63
accept_6, 63
accept_7, 64
autobind, 37

badf_1, 203
bind_1, 67
bind_2, 68
bind_3, 68
bind_5, 69
bind_7, 69
bind_9, 70
both_streams_destroyed, 264
bound_after, 37
bound_port_allowed, 36
bound_ports_protocol_autobind, 36

calculate_buf_sizes, 40
calculate_tcp_options_len, 40
call, 270
close_1, 72
close_10, 80
close_2, 73
close_3, 74
close_4, 75
close_5, 76
close_6, 77
close_7, 77
close_8, 78
connect_1, 84
connect_10, 98
connect_1a, 86
connect_2, 89
connect_3, 89
connect_4, 90
connect_4a, 91
connect_5, 91
connect_5a, 92
connect_5b, 93
connect_5c, 94
connect_5d, 95
connect_6, 95
connect_7, 96
connect_8, 97
connect_9, 97

deliver_in_1, 208
deliver_in_2, 210
deliver_in_3, 212
deliver_in_3b, 220
deliver_in_4, 221
deliver_in_5, 221
deliver_in_7, 222
deliver_in_7a, 223
deliver_in_7b, 224
deliver_in_7c, 224
deliver_in_7d, 225
deliver_in_8, 226
deliver_in_9, 226
deliver_in_99, 249
deliver_in_99a, 250
deliver_in_icmp_1, 241
deliver_in_icmp_2, 242
deliver_in_icmp_3, 244
deliver_in_icmp_4, 245
deliver_in_icmp_5, 246
deliver_in_icmp_6, 246
deliver_in_icmp_7, 247
deliver_in_udp_1, 239
deliver_in_udp_2, 240
deliver_in_udp_3, 240
deliver_loop_99, 250
deliver_out_1, 230
deliver_out_99, 250
destroy, 264
destroy_quads, 265
di3_ackstuff, 215
di3_datastuff, 216
di3_newackstuff, 214
di3_socks_update, 219
di3_ststuff, 216
di3_topstuff, 214
disconnect_1, 102
disconnect_2, 103
disconnect_3, 103
disconnect_4, 100
disconnect_5, 101
do_tcp_options, 40
dosend, 42
dropwithreset, 51
dup_1, 105
dup_2, 105

dupfd_1, 107
dupfd_3, 107
dupfd_4, 108

enqueue_and_ignore_fail, 50
enqueue_each_and_ignore_fail, 50
enqueue_or_fail, 50
ephemeral_ports, 21
ERROR, 277
exists_quad_of, 53

fmap_every, 256
fmap_every_pred, 256

getfileflags_1, 109
getifaddrs_1, 111
getpeername_1, 114
getpeername_2, 114
getsockbopt_1, 117
getsockbopt_2, 117
getsockerr_1, 119
getsockerr_2, 119
getsocklistening_1, 121
getsocklistening_2, 122
getsocklistening_3, 122
getsockname_1, 124
getsockname_2, 125
getsockname_3, 125
getsocknopt_1, 127
getsocknopt_4, 128
getsocktopt_1, 130
getsocktopt_4, 130

host, 21
host1_to_3, 276
host_tau, 271

initial_cb, 43
initial_stream, 261
initial_streamFlags, 261
initial_streams, 262
interface, 271
interface_1, 254
intr_1, 203

Lhost0, 31
listen_1, 132
listen_1b, 133
listen_1c, 134
listen_2, 134
listen_3, 135
listen_4, 136
listen_5, 136
listen_7, 137
Lnet0, 269
lookup_icmp, 39
lookup_udp, 38

make_syn_ack_flgs_data, 263
make_syn_flgs_data, 262

match_score, 37
mtu_tab, 42

next_smaller, 42
notsock_1, 203
null_flgs_data, 262

privileged_ports, 21
proto_eq, 20
proto_of, 20
protocol_info, 20

quad_of, 53

read, 263
recv_1, 139
recv_11, 150
recv_12, 151
recv_13, 152
recv_14, 152
recv_15, 153
recv_16, 153
recv_17, 154
recv_2, 141
recv_20, 155
recv_21, 156
recv_22, 157
recv_23, 157
recv_24, 158
recv_3, 142
recv_4, 143
recv_7, 144
recv_8, 144
recv_8a, 145
recv_9, 146
remove_destroyed_streams, 264
resourcefail_1, 204
resourcefail_2, 205
return, 270
return_1, 202
rn, 270
rollback_tcp_output, 48

sane_socket, 36
send_1, 161
send_10, 174
send_11, 175
send_12, 176
send_13, 177
send_14, 178
send_15, 179
send_16, 179
send_17, 180
send_18, 181
send_19, 182
send_2, 163
send_21, 182
send_22, 183
send_23, 184

send_3, 5, 7, 163
send_3a, 164
send_4, 165
send_5, 166
send_5a, 166
send_6, 167
send_7, 167
send_8, 168
send_9, 173
send_queue_space, 41
setfileflags_1, 185
setsockbopt_1, 187
setsockbopt_2, 188
setsocknopt_1, 190
setsocknopt_2, 191
setsocknopt_4, 191
setsocktopt_1, 193
setsocktopt_4, 194
setsocktopt_5, 194
shutdown_1, 196
shutdown_2, 197
shutdown_3, 198
shutdown_4, 198
Sock, 20
socket, 20
socket1_to_3, 275
socket_1, 200
socket_2, 201
soreadable, 258
sowriteable, 257
stream_enqueue_or_fail, 50
stream_enqueue_or_fail_sock, 50
stream_loopback_on_wire, 35
stream_mlift_dropafterack_or_fail, 52
stream_mlift_tcp_output_perhaps_or_fail, 50
stream_reass, 276
stream_rollback_tcp_output, 49
stream_tcp_output_perhaps, 48
stream_tcp_output_really, 47
stream_test_outroute, 35
streamFlags, 6, 27
streamid_of_quad, 262
sync_streams, 263

tau, 270
tcp_close, 52
tcp_drop_and_close, 53
tcp_output_perhaps, 48
tcp_output_really, 45
tcp_output_required, 45
TCP_Sock, 20
TCP_Sock0, 20
tcp_sock_of, 20
tcp_socket, 20
tcp_socket1_to_3, 275
tcp_socket_best_match, 38
tcpcb, 19
tcpcb1_to_3, 275
tcpStream, 6, 27
tcpStreams, 28
time_pass, 271
Time_Pass_host, 256
Time_Pass_socket, 256
Time_Pass_tcpcb, 256
Time_Pass_timedoption, 255
timer_tt_2msl_1, 237
timer_tt_conn_est_1, 237
timer_tt_delack_1, 237
timer_tt_fin_wait_2_1, 238
timer_tt_keep_1, 236
timer_tt_persist_1, 236
timer_tt_rexmt_1, 234
timer_tt_rexmtsyn_1, 233
trace, 271
trace_1, 253
trace_2, 253
tracecb_eq, 22
tracesock_eq, 22
type_abbrev_hosts, 269
type_abbrev_msgs, 269
type_abbrev_net, 269
type_abbrev_streamid, 27
type_abbrev_streams, 269
type_abbrev_tracerecord, 22

UDP_Sock, 20
UDP_Sock0, 20
udp_sock_of, 20
update_idle, 51

write, 6, 263