Technical Report

UCAM-CL-TR-625
ISSN 1476-2986

Number 625

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

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

March 2005

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: rigorous and experimentally-validated behavioural specification

Volume 2: The Specification

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

*University of Cambridge Computer Laboratory
†NICTA, Canberra

March 18, 2005

Brief Contents

How to read this document

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

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

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

Full Contents

Part I

TCP1_utils

Chapter 1

Utility functions

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

1.1 Basic utilities

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

1.1.1 Summary

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

1.1.2 Rules

– **update one point of a function:**
$f \oplus (x \mapsto y) = \lambda x'.\mathbf{if}\ \ x' = x\ \ \mathbf{then}\ \ y\ \ \mathbf{else}\ \ f\ x'$

– **update multiple points of a function:**
$\mathrm{funupd\_list}\ \ f\ \ xys = \mathbf{foldl}(\lambda f(x, y).f \oplus (x \mapsto y))f\ \ xys$

– **clip int to num :**
$\mathrm{clip\_int\_to\_num}(i : \mathsf{int}) = \mathbf{if}\ \ i < 0\ \ \mathbf{then}\ \ 0\ \ \mathbf{else}\ \ \mathbf{num}\ \ i$

– **left shift, written $\ll$:**
$\mathrm{left\_shift\_num}(n : \mathsf{num})(i : \mathsf{num}) = n * 2\ \ ^{**}\ \ i$

– **right shift, written $\gg$:**
$\mathrm{right\_shift\_num}(n : \mathsf{num})(i : \mathsf{num}) = n\ \ \mathbf{div}\ \ 2\ \ ^{**}\ \ i$

– **round v down to multiple of bs, unless $v < bs$ already :**
$\mathrm{rounddown}\ \ bs\ \ v = \mathbf{if}\ \ v < bs\ \ \mathbf{then}\ \ v\ \ \mathbf{else}\ (v\ \ \mathbf{div}\ \ bs) * bs$

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

roundup $bs\ v = ((v + (bs - 1))\ \mathbf{div}\ bs) * bs$

– **inject** int **into** *real* **:**
$\mathrm{real\_of\_int}(i : \mathsf{int}) = \mathbf{if}\ \ i < 0\ \ \mathbf{then}\ \neg(\mathbf{real\_of\_num}(\mathbf{num}\,\neg i))$
$\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{real\_of\_num}(\mathbf{num}\ \ i)$

– num **floor of** *real* **:**
$\mathrm{num\_floor}(x : real) = \mathbf{least}(n : \mathsf{num}).\,\mathbf{real\_of\_num}(n + 1) > x$

– num **floor and fractional part of** *real* **:**
$\mathrm{num\_floor\_and\_frac}(x : real)$
$= \mathbf{let}\ \ n = \mathbf{least}(n : \mathsf{num}).\,\mathbf{real\_of\_num}(n + 1) > x$
$\mathbf{in}$
$(n, x - \mathbf{real\_of\_num}\ \ n)$

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

– **used for conditional record updates :**
$(x\ \ \mathbf{onlywhen}\ \ b) = \mathbf{if}\ \ b\ \ \mathbf{then}\ \ K\ x\ \ \mathbf{else}\ \ \mathbf{I}$

1.2 List utilities

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

1.2.1 Summary

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

1.2.2 Rules

– **split worker function:**
$(\mathrm{SPLIT\_REV\_0}\ \ 0\ ls\ rs = (ls, rs)) \wedge$
$(\mathrm{SPLIT\_REV\_0}(SUC\ \ n)ls(r :: rs) = \mathrm{SPLIT\_REV\_0}\ \ n(r :: ls)rs) \wedge$
$(\mathrm{SPLIT\_REV\_0}(SUC\ \ n)ls[\,] = (ls, [\,]))$

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

SPLIT_REV $n\ rs$ = SPLIT_REV_0 $n[\,]rs$

– **split a list after n elements, returning the prefix and the remainder:**
SPLIT $n\ rs$ = **let** (ls, rs) = SPLIT_REV $n\ rs$ **in** $(\mathit{REVERSE}\ ls, rs)$

– **take the first n elements of a list:**
TAKE $n\ rs$ = **let** (ls, rs) = SPLIT_REV $n\ rs$ **in** $\mathit{REVERSE}\ ls$

– **drop the first n elements of a list:**
DROP $n\ rs$ = **let** (ls, rs) = SPLIT_REV $n\ rs$ **in** rs

– **split a list at first element not satisfying p, returning reversed prefix and remainder:**
TAKEWHILE_REV $p\ ls(r :: rs)$ = TAKEWHILE_REV $p($**if** $p\ r$ **then** $(r :: ls)$ **else** $ls)rs \wedge$
TAKEWHILE_REV $p\ ls[\,] = ls$

– **split a list at first element not satisfying p, returning prefix and remainder:**
TAKEWHILE $p\ rs = \mathit{REVERSE}($TAKEWHILE_REV $p[\,]rs)$

– **make a list of n copies of x:**
(REPLICATE $0\ x = [\,]) \wedge$
(REPLICATE$(\mathit{SUC}\ n)x = x ::$ REPLICATE $n\ x)$

– **decrement a list of nums by a num, dropping any that count below zero:**
((decr_list : num → num list → num list)
$d[\,] = [\,]) \wedge$
(decr_list $d(n :: ns)$ = (**if** $n < d$ **then** **I** **else** $\mathit{CONS}(n - d))$(decr_list $d\ ns$))

– **not in :**
$(x \notin y) = \neg($**mem** $x\ y)$

– **map with optional result:**
MAP_OPTIONAL $f(x :: xs)$ = **append**(**case** $f\ x$ **of**
$* \to [\,]$
$\| \uparrow y \to [y])$
(MAP_OPTIONAL $f\ xs) \wedge$
MAP_OPTIONAL $f[\,] = [\,]$

– **concatentation of option list that drops all $*$s:**
CONCAT_OPTIONAL xs = MAP_OPTIONAL **I** xs

– **the set of all orderings of a set :**
ORDERINGS $s\ l$ = (**list_to_set** $l = s \wedge$
length l = **card** s)

– **insert ordered:**
INSERT_ORDERED $new\ old\ bad$ =
filter$(\lambda fd.fd \in new \vee fd \in bad)old$

1.3 Assertions

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

1.3.1 Summary

ASSERTION_FAILURE	assertion failure (causes checker to halt)

1.3.2 Rules

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

$ASSERTION\_FAILURE(s : \mathsf{string}) = \mathbf{F}$

Part II

TCP1_errors

Chapter 2

Error codes

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

2.1 The type of errors

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

2.1.1 Summary

error

2.1.2 Rules

– :
error =
  E2BIG
| EACCES
| EADDRINUSE
| EADDRNOTAVAIL
| EAFNOSUPPORT
| EAGAIN
| EWOULDBLOCK (* only used if EWOULDBLOCK $\neq$ EAGAIN *)
| EALREADY
| EBADF
| EBADMSG
| EBUSY
| ECANCELED
| ECHILD
| ECONNABORTED
| ECONNREFUSED
| ECONNRESET
| EDEADLK
| EDESTADDRREQ
| EDOM
| EDQUOT
| EEXIST
| EFAULT
| EFBIG
| EHOSTUNREACH

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

Part III

TCP1_signals

Chapter 3

Signal names

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

3.1 The type of signals

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

3.1.1 Summary

signal

3.1.2 Rules

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

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

Part IV

TCP1_baseTypes

Chapter 4

Base types

This file defines basic types used throughout the specification.

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

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

4.1.1 Summary

port
ip
ifid
netmask
fd

4.1.2 Rules

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

Description TCP or UDP port number, non-zero.

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

Description IPv4 address, non-zero.

– :
ifid = LO | ETH **of** num

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

– :
netmask = NETMASK **of** num

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

– :
fd = FD **of** num

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

4.2 File and socket flags (TCP and UDP)

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

4.2.1 Summary

filebflag
sockbflag
socknflag
socktflag
msgbflag
socktype

4.2.2 Rules

– :
filebflag = O_NONBLOCK
| O_ASYNC

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

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

| SO_OOBINLINE(* ? *)
| SO_DONTROUTE

Description Boolean flags affecting the behaviour of a socket.

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

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

SO_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.

Variations

Linux	The flag SO_BSDCOMPAT is Linux-only.

– :
socknflag = SO_SNDBUF
| SO_RCVBUF
| SO_SNDLOWAT
| SO_RCVLOWAT

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

SO_SNDBUF Specifies the send buffer size.

SO_RCVBUF Specifies the receive buffer size.

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

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

– :
socktflag = SO_LINGER
| SO_SNDTIMEO
| SO_RCVTIMEO

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

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

SO_RCVTIMEO specifies the timeout value for input operations.

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

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

Description Boolean flags affecting the behaviour of a send or recv call.
MSG_DONTWAIT: Do not block if there is no data available.
MSG_OOB: Return out-of-band data.
MSG_PEEK: Read data but do not remove it from the socket's receive queue.
MSG_WAITALL: Block untill all n bytes of data are available.

– :
socktype = SOCK_STREAM
| SOCK_DGRAM

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

4.3 Language interaction types

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

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

4.3.1 Summary

tid
err
TLang_type
TLang
tlang_typing

4.3.2 Rules

– :
tid = TID **of** num

Description Thread IDs.

– :
err = OK **of** $'a$ | FAIL **of** error

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

– :

$\mathsf{TLang\_type} = \mathrm{TL}\textsc{ty\_int}$
| $\mathrm{TL}\textsc{ty\_bool}$
| $\mathrm{TL}\textsc{ty\_string}$
| $\mathrm{TL}\textsc{ty\_one}$
| $\mathrm{TL}\textsc{ty\_pair}$ **of** $(\mathsf{TLang\_type} \# \mathsf{TLang\_type})$
| $\mathrm{TL}\textsc{ty\_list}$ **of** $\mathsf{TLang\_type}$
| $\mathrm{TL}\textsc{ty\_lift}$ **of** $\mathsf{TLang\_type}$
| $\mathrm{TL}\textsc{ty\_err}$ **of** $\mathsf{TLang\_type}$
| $\mathrm{TL}\textsc{ty\_fd}$
| $\mathrm{TL}\textsc{ty\_ip}$
| $\mathrm{TL}\textsc{ty\_port}$
| $\mathrm{TL}\textsc{ty\_error}$
| $\mathrm{TL}\textsc{ty\_netmask}$
| $\mathrm{TL}\textsc{ty\_ifid}$
| $\mathrm{TL}\textsc{ty\_filebflag}$
| $\mathrm{TL}\textsc{ty\_sockbflag}$
| $\mathrm{TL}\textsc{ty\_socknflag}$
| $\mathrm{TL}\textsc{ty\_socktflag}$
| $\mathrm{TL}\textsc{ty\_socktype}$
| $\mathrm{TL}\textsc{ty\_tid}$
| $\mathrm{TL}\textsc{ty\_signal}$

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

– :

$\mathsf{TLang} = \mathrm{TL}\textsc{\_int}$ **of** int
| $\mathrm{TL}\textsc{\_bool}$ **of** bool
| $\mathrm{TL}\textsc{\_string}$ **of** string
| $\mathrm{TL}\textsc{\_one}$ **of** $()$
| $\mathrm{TL}\textsc{\_pair}$ **of** $\mathsf{TLang} \# \mathsf{TLang}$
| $\mathrm{TL}\textsc{\_list}$ **of** $\mathsf{TLang\ list}$
| $\mathrm{TL}\textsc{\_option}$ **of** $\mathsf{TLang\ option}$
| $\mathrm{TL}\textsc{\_err}$ **of** $\mathsf{TLang\ err}$
| $\mathrm{TL}\textsc{\_fd}$ **of** fd
| $\mathrm{TL}\textsc{\_ip}$ **of** ip
| $\mathrm{TL}\textsc{\_port}$ **of** port
| $\mathrm{TL}\textsc{\_error}$ **of** error
| $\mathrm{TL}\textsc{\_netmask}$ **of** $\mathit{netmask}$
| $\mathrm{TL}\textsc{\_ifid}$ **of** ifid
| $\mathrm{TL}\textsc{\_filebflag}$ **of** $\mathsf{filebflag}$
| $\mathrm{TL}\textsc{\_sockbflag}$ **of** $\mathsf{sockbflag}$
| $\mathrm{TL}\textsc{\_socknflag}$ **of** $\mathsf{socknflag}$
| $\mathrm{TL}\textsc{\_socktflag}$ **of** $\mathsf{socktflag}$
| $\mathrm{TL}\textsc{\_socktype}$ **of** $\mathsf{socktype}$
| $\mathrm{TL}\textsc{\_tid}$ **of** tid
| $\mathrm{TL}\textsc{\_signal}$ **of** signal

Description Language values.

– :

$(\forall i.\mathit{tlang\_typing}(\mathrm{TL}\textsc{\_int}\ i)\mathrm{TL}\textsc{ty\_int}) \wedge$

$(\forall b. \mathit{tlang\_typing}(\text{TL\_BOOL } b)\text{TLTY\_BOOL}) \wedge$

$(\forall s. \mathit{tlang\_typing}(\text{TL\_STRING } s)\text{TLTY\_STRING}) \wedge$

$\mathit{tlang\_typing}(\text{TL\_ONE } ())\text{TLTY\_ONE} \wedge$

$(\forall p_1\ p_2\ ty_1\ ty_2.$
$\quad \mathit{tlang\_typing}\ p_1\ ty_1 \wedge \mathit{tlang\_typing}\ p_2\ ty_2 \implies$
$\quad \mathit{tlang\_typing}(\text{TL\_PAIR}(p_1, p_2))(\text{TLTY\_PAIR}(ty_1, ty_2))) \wedge$

$(\forall tl\ ty.(\forall e. \mathbf{mem}\ e\ tl \implies \mathit{tlang\_typing}\ e\ ty) \implies$
$\quad \mathit{tlang\_typing}(\text{TL\_LIST } tl)(\text{TLTY\_LIST } ty)) \wedge$

$(\forall p\ ty. \mathit{tlang\_typing}\ p\ ty \implies$
$\quad \mathit{tlang\_typing}(\text{TL\_OPTION}(\uparrow p))(\text{TLTY\_LIFT } ty)) \wedge$
$(\forall ty. \mathit{tlang\_typing}(\text{TL\_OPTION } *)(\text{TLTY\_LIFT } ty)) \wedge$

$(\forall e\ ty. \mathit{tlang\_typing}(\text{TL\_ERR}(\text{FAIL } e))(\text{TLTY\_ERR } ty)) \wedge$
$(\forall p\ ty. \mathit{tlang\_typing}\ p\ ty \implies$
$\quad \mathit{tlang\_typing}(\text{TL\_ERR}(\text{OK } p))(\text{TLTY\_ERR } ty)) \wedge$

$(\forall fd. \mathit{tlang\_typing}(\text{TL\_FD } fd)\text{TLTY\_FD}) \wedge$

$(\forall i. \mathit{tlang\_typing}(\text{TL\_IP } i)\text{TLTY\_IP}) \wedge$
$(\forall p. \mathit{tlang\_typing}(\text{TL\_PORT } p)\text{TLTY\_PORT}) \wedge$
$(\forall e. \mathit{tlang\_typing}(\text{TL\_ERROR } e)\text{TLTY\_ERROR}) \wedge$
$(\forall nm. \mathit{tlang\_typing}(\text{TL\_NETMASK } nm)\text{TLTY\_NETMASK}) \wedge$
$(\forall ifid. \mathit{tlang\_typing}(\text{TL\_IFID } ifid)\text{TLTY\_IFID}) \wedge$
$(\forall ff. \mathit{tlang\_typing}(\text{TL\_FILEBFLAG } ff)\text{TLTY\_FILEBFLAG}) \wedge$
$(\forall sf. \mathit{tlang\_typing}(\text{TL\_SOCKBFLAG } sf)\text{TLTY\_SOCKBFLAG}) \wedge$
$(\forall sf. \mathit{tlang\_typing}(\text{TL\_SOCKNFLAG } sf)\text{TLTY\_SOCKNFLAG}) \wedge$
$(\forall sf. \mathit{tlang\_typing}(\text{TL\_SOCKTFLAG } sf)\text{TLTY\_SOCKTFLAG}) \wedge$
$(\forall st. \mathit{tlang\_typing}(\text{TL\_SOCKTYPE } st)\text{TLTY\_SOCKTYPE}) \wedge$
$(\forall tid. \mathit{tlang\_typing}(\text{TL\_TID } tid)\text{TLTY\_TID}) \wedge$
(* (!l ty. tlang_typing (TL_ref (Loc (ty,l))) (TLty_ref ty)) /\ *)
(* (!ex. tlang_typing (TL_exn ex) TLty_exn) /\ *)
(* (!p ty. tlang_typing p ty ==> *)
(* tlang_typing (TL_except (EOK p)) (TLty_except ty)) /\ *)
(* (!ex ty. tlang_typing (TL_exn ex) TLty_exn ==> *)
(* tlang_typing (TL_except (EEX ex)) (TLty_except ty)) /\ *)
$(\forall s. \mathit{tlang\_typing}(\text{TL\_SIGNAL } s)\text{TLTY\_SIGNAL})$

4.4 Time types

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

4.4.1 Summary

time	
type_abbrev_duration	
time_lt	written $<$
time_lte	written $\leq$
time_gt	written $>$
time_gte	written $\geq$

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

4.4.2 Rules

– :
time $= \infty$ | time **of** *real*

– :
type_abbrev duration : *real*

– written $<$:
$((\text{time\_lt} : \mathsf{time} \to \mathsf{time} \to \mathsf{bool})(\mathsf{time}\ x)(\mathsf{time}\ y) = x < y)$
$\wedge (\text{time\_lt}\ \infty\ ys = \mathbf{F})$
$\wedge (\text{time\_lt}\ xs\ \infty = \mathbf{T})$

– written $\leq$:
$\text{time\_lte}(\mathsf{time}\ x)(\mathsf{time}\ y) = x \leq y \wedge$
$\text{time\_lte}\ t\ \infty = \mathbf{T} \wedge$
$\text{time\_lte}\ \infty\ t = (t = \infty)$

– written $>$:
$\text{time\_gt}\ xs\ ys = \text{time\_lt}\ ys\ xs$

– written $\geq$:
$\text{time\_gte}\ xs\ ys = \text{time\_lte}\ ys\ xs$

– written min x y :
$\text{time\_min}(\mathsf{time}\ x)(\mathsf{time}\ y) = \mathsf{time}(\mathbf{min}\ x\ y) \wedge$
$\text{time\_min}(\mathsf{time}\ x)\infty = \mathsf{time}\ x \wedge$
$\text{time\_min}\ \infty(\mathsf{time}\ x) = \mathsf{time}\ x \wedge$
$\text{time\_min}\ \infty\ \infty = \infty$

– written max x y :
$\text{time\_max}(\mathsf{time}\ x)(\mathsf{time}\ y) = \mathsf{time}(\mathbf{max}\ x\ y) \wedge$
$\text{time\_max}\ \infty(\mathsf{time}\ x) = \infty \wedge$
$\text{time\_max}(\mathsf{time}\ x)\infty = \infty \wedge$
$\text{time\_max}\ \infty\ \infty = \infty$

– written $+$:
$((\text{time\_plus\_dur} : \mathsf{time} \to \text{duration} \to \mathsf{time})$
$(\mathsf{time}\ x)y = \mathsf{time}(x + y)) \wedge$

$(\text{time\_plus\_dur}\ \infty\ y = \infty)$

– written – :

$((\text{time\_minus\_dur} : \mathsf{time} \to \text{duration} \to \mathsf{time})$
$(\mathsf{time}\ x)y = \mathsf{time}(x - y)) \wedge$
$(\text{time\_minus\_dur}\ \infty\ y = \infty)$

– written ∗ :

$(\text{real\_mult\_time} : \mathit{real} \to \mathsf{time} \to \mathsf{time})$
$x(\mathsf{time}\ y) = \mathsf{time}(x * y) \wedge$
$\text{real\_mult\_time}\ x\ \infty = \infty$

– :

$(0 : \mathsf{time}) = \mathsf{time}\ 0$

– :

$(\text{duration} : \mathsf{num} \to \mathsf{num} \to \text{duration})\mathit{sec}\ \mathit{usec} = \$\&\mathit{sec} + \$\&\mathit{usec}/1000000$

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

– :

$(\text{abstime} : \mathsf{num} \to \mathsf{num} \to \text{duration})\mathit{sec}\ \mathit{usec} = \$\&\mathit{sec} + \$\&\mathit{usec}/1000000$

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

– :

$(\text{realopt\_of\_time} : \mathsf{time} \to \mathit{real}\ \ \mathsf{option})(\mathsf{time}\ x) = \uparrow x \wedge$
$\text{realopt\_of\_time}\ \infty = *$

– written the :

$\text{the\_time}(\mathsf{time}\ x) = x$

4.5 Basic network types: sequence numbers (TCP only)

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

4.5.1 Summary

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

4.5.2 Rules

– :
type_abbrev *byte* : *char*

– :
$seq_{32} = \mathrm{SEQ32}$ **of** $'a => word32$

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

– written + :
$\mathrm{seq32\_plus}(\mathrm{SEQ32}\ a\ n)(m : \mathsf{num}) = \mathrm{SEQ32}\ a(n + \mathbf{n2w}\ m)$

– written − :
$\mathrm{seq32\_minus}(\mathrm{SEQ32}\ a\ n)(m : \mathsf{num}) = \mathrm{SEQ32}\ a(n - \mathbf{n2w}\ m)$

– written + :
$\mathrm{seq32\_plus}'(\mathrm{SEQ32}\ a\ n)(m : \mathsf{int}) = \mathrm{SEQ32}\ a(n + i2w\ m)$

– written − :
$\mathrm{seq32\_minus}'(\mathrm{SEQ32}\ a\ n)(m : \mathsf{int}) = \mathrm{SEQ32}\ a(n - i2w\ m)$

– written − :

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

– **written** $<$ **:**
seq32_lt(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = $((n - m) : \mathsf{int}) < 0$

– **written** $\leq$ **:**
seq32_leq(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = $((n - m) : \mathsf{int}) \leq 0$

– **written** $>$ **:**
seq32_gt(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = $((n - m) : \mathsf{int}) > 0$

– **written** $\geq$ **:**
seq32_geq(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = $((n - m) : \mathsf{int}) \geq 0$

– **:**
seq32_fromto(a : $'a$)b(SEQ32(c : $'a$)n) = SEQ32 b n

– **:**
seq32_coerce(SEQ32 a n) = SEQ32 ARB n

– **written min** x y **:**
seq32_min(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = **if** $n < m$ **then** n **else** m

– **written max** x y **:**
seq32_max(n : $'a$ seq_{32})(m : $'a$ seq_{32}) = **if** $n < m$ **then** m **else** n

– **:**
tcpLocal = TcpLocal

– **:**
tcpForeign = TcpForeign

– **:**
type_abbrev tcp_seq_local : tcpLocal seq_{32}

– **:**
type_abbrev tcp_seq_foreign : tcpForeign seq_{32}

– **:**
tcp_seq_local(n : $word32$) = SEQ32 TcpLocal n

– **:**
tcp_seq_foreign(n : $word32$) = SEQ32 TcpForeign n

– **:**
tcp_seq_local_to_foreign = seq32_coerce : tcp_seq_local → tcp_seq_foreign

– **:**
tcp_seq_foreign_to_local = seq32_coerce : tcp_seq_foreign → tcp_seq_local

– :
$\mathsf{tstamp} = \textsc{Tstamp}$

– :
type_abbrev ts_seq : $\mathsf{tstamp}\ seq_{32}$

– :
$\mathrm{ts\_seq}(n : \mathit{word32}) = \mathrm{SEQ32}\ \textsc{Tstamp}\ n$

Part V

TCP1_netTypes

Chapter 5

Network datagram types

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

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

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

```
3.1.  Internet Header Format

  A summary of the contents of the internet header follows:

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Version|  IHL  |Type of Service|          Total Length         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Identification        |Flags|      Fragment Offset    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Time to Live |    Protocol   |         Header Checksum       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Source Address                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Destination Address                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


  TCP Header Format

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

5.1 TCP segments (TCP only)

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

5.1.1 Summary

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

5.1.2 Rules

– **TCP datagram type :**
tcpSegment
=⟨| is_1 : ip option; (* source IP *)
  is_2 : ip option; (* destination IP *)
  ps_1 : port option; (* source port *)
  ps_2 : port option; (* destination port *)
  seq : tcp_seq_local; (* sequence number *)
  ack : tcp_seq_foreign; (* acknowledgment number *)
  URG : bool;
  ACK : bool;
  PSH : bool;
  RST : bool;
  SYN : bool;
  FIN : bool;
  win : *word16*; (* window size (unsigned) *)
  ws : *byte* option; (* TCP option: window scaling; typically 0..14 *)

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

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

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

5.2 UDP datagrams (UDP only)

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

5.2.1 Summary

udpDatagram	UDP datagram type
sane_udpdgm	message well-formedness test (physical constraints imposed by format)

5.2.2 Rules

– **UDP datagram type :**
udpDatagram
=⟨| is_1 : ip option; (* source IP *)
is_2 : ip option; (* destination IP *)
ps_1 : port option; (* source port *)
ps_2 : port option; (* destination port *)
data : *byte* list
|⟩

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

5.3 ICMP datagrams (TCP and UDP)

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

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

5.3.1 Summary

protocol	protocol type for use in ICMP messages
icmp_unreach_code	
icmp_source_quench_code	
icmp_redirect_code	
icmp_time_exceeded_code	
icmp_paramprob_code	
icmpType	
icmpDatagram	ICMP datagram type

5.3.2 Rules

– **protocol type for use in ICMP messages :**
protocol = PROTO_TCP | PROTO_UDP

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

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

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

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

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

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

– **ICMP datagram type :**
icmpDatagram
 =⟪ is_1 : ip option; (* this is the sender of this ICMP *)
 is_2 : ip option; (* this is the intended receiver of this ICMP *)

 (* we assume the enclosed IP always has at least 8 bytes of data, i.e., enough for all the fields below *)
 is_3 : ip option; (* source of enclosed IP datagram *)
 is_4 : ip option; (* destination of enclosed IP datagram *)
 ps_3 : port option; (* source port *)
 ps_4 : port option; (* destination port *)
 proto : protocol; (* protocol *)
 seq : tcp_seq_local option; (* seq *)
 t : icmpType
 ⟫

5.4 IP messages (TCP and UDP)

An IP datagram is (for our purposes) either a TCP segment, an ICMP datagram, or a UDP datagram. We use the type msg for IP datagrams. IP datagrams may be checked for sanity, and may have their is_1 and is_2 fields inspected.

5.4.1 Summary

msg	IP message type
sane_msg	message well-formedness test (physical constraints imposed by format)
msg_is1	source IP of a message, written $x.is_1$
msg_is2	destination IP of a message, written $x.is_2$

5.4.2 Rules

– **IP message type :**
msg = TCP **of** tcpSegment | ICMP **of** icmpDatagram | UDP **of** udpDatagram

– **message well-formedness test (physical constraints imposed by format) :**
sane_msg(TCP *seg*) = sane_seg *seg* $\wedge$
sane_msg(ICMP *dgm*) = **T** $\wedge$
sane_msg(UDP *dgm′*) = sane_udpdgm *dgm′*

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

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

Part VI

TCP1_LIBinterface

Chapter 6

System call types

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

6.1 The interface (TCP and UDP)

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

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

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

6.1.1 Summary

LIB_interface
retType

6.1.2 Rules

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

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

Description Sockets calls with their argument types.

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

$\wedge$ retType(socket _) = TLTY_FD

Description Return types of sockets calls.

6.2 Useful groups of calls (TCP and UDP)

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

6.2.1 Summary

fd_op
fd_sockop

6.2.2 Rules

– :
fd_op fd opn = (
opn = accept(fd) $\vee$
$(\exists is\ ps.opn = \text{bind}(\text{fd}, is, ps)) \vee$
opn = close(fd) $\vee$
$(\exists i\ p.opn = \text{connect}(\text{fd}, i, p)) \vee$
opn = disconnect(fd) $\vee$
opn = dup(fd) $\vee$
$(\exists fd'.opn = \text{dupfd}(\text{fd}, fd')) \vee$
$(opn = \text{getfileflags}(\text{fd})) \vee$
$(\exists flags.opn = \text{setfileflags}(\text{fd}, flags)) \vee$
opn = getsockname(fd) $\vee$
opn = getpeername(fd) $\vee$
$(\exists sfb.opn = \text{getsockbopt}(\text{fd}, sfb)) \vee$
$(\exists sfn.opn = \text{getsocknopt}(\text{fd}, sfn)) \vee$
$(\exists sft.opn = \text{getsocktopt}(\text{fd}, sft)) \vee$
$(\exists sfb\ b.opn = \text{setsockbopt}(\text{fd}, sfb, b)) \vee$
$(\exists sfn\ n.opn = \text{setsocknopt}(\text{fd}, sfn, n)) \vee$
$(\exists sft\ t.opn = \text{setsocktopt}(\text{fd}, sft, t)) \vee$
$(\exists n.opn = \text{listen}(\text{fd}, n)) \vee$
$(\exists n\ opt.opn = \text{recv}(\text{fd}, n, opt)) \vee$
$(\exists data\ opt.opn = \text{send}(\text{fd}, data, opt)) \vee$
$(\exists r\ w.opn = \text{shutdown}(\text{fd}, r, w)) \vee$
opn = sockatmark(fd) $\vee$
opn = getsockerr(fd) $\vee$
opn = getsocklistening(fd)
)

Description Calls that expect a (single) fd.

– :
fd_sockop fd opn = (
opn = accept(fd) $\vee$

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

Description Calls that expect a (single) socket fd.

Part VII

TCP1_host0

Chapter 7

Host LTS labels and rule categories

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

7.1 Transition labels (TCP and UDP)

Host transition labels.

7.1.1 Summary

Lhost0	Host transition labels

7.1.2 Rules

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

(* message transmission and receipt *)
| LH_SENDDATAGRAM **of** msg (* output of message to the network, written $\overline{\mathsf{msg}}$ *)
| LH_RECVDATAGRAM **of** msg (* input of message from the network, written msg *)
| LH_LOOPDATAGRAM **of** msg (* loopback output/input, written $\overleftrightarrow{\mathsf{msg}}$ *)

(* 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_EPSILON **of** duration (* time passage, written *dur* *)
| LH_TRACE **of** *tracerecord* (* TCP trace record, written LH_TRACE *tr* *)

7.2 Rule categories (TCP and UDP)

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

7.2.1 Summary

rule_proto
rule_status
rule_cat
urgent
nonurgent
is_urgent

7.2.2 Rules

– :
rule_proto = RP_TCP
| RP_UDP
| RP_ALL

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

– :
rule_status = SUCCEED
| FAIL
| BADFAIL

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

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

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

$-$ **:**
urgent $=$ **T**

$-$ **:**
nonurgent $=$ **F**

$-$ **:**
is_urgent(SLOW b _) $= b \wedge$
is_urgent(NETWORK b) $= b \wedge$
is_urgent(MISC b) $= b \wedge$
is_urgent _ $=$ **F**

Part VIII

TCP1_ruleids

Chapter 8

Rule names

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

8.1 names (Rule only)

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

8.1.1 Summary

rule_ids

8.1.2 Rules

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

| *recv_23* | *recv_24*
| *send_1* | *send_2* | *send_3* | *send_3a* | *send_4* | *send_5* | *send_5a*
| *send_6* | *send_7* | *send_8* | *send_9* | *send_10*
| *send_11* | *send_12* | *send_13* | *send_14* | *send_15*
| *send_16* | *send_17* | *send_18* | *send_19* | *send_21* | *send_22* | *send_23*
| *sockatmark_1* | *sockatmark_2*
| *pselect_1* | *pselect_2* | *pselect_3* | *pselect_4* | *pselect_5*
| *pselect_6*
| *getsockname_1* | *getsockname_2* | *getsockname_3*
| *getpeername_1* | *getpeername_2*
| *badf_1*
| *notsock_1*
| *intr_1*
| *resourcefail_1* | *resourcefail_2*
| *deliver_in_1* | *deliver_in_1b* | *deliver_in_2* | *deliver_in_2a*
| *deliver_in_3* | *deliver_in_3a* | *deliver_in_3b* | *deliver_in_3c*
| *deliver_in_4* | *deliver_in_5* | *deliver_in_6*
| *deliver_in_7* | *deliver_in_7a* | *deliver_in_7b* | *deliver_in_7c*
| *deliver_in_7d* | *deliver_in_8* | *deliver_in_9*
| *deliver_in_icmp_1* | *deliver_in_icmp_2* | *deliver_in_icmp_3*
| *deliver_in_icmp_4* | *deliver_in_icmp_5* | *deliver_in_icmp_6*
| *deliver_in_icmp_7*
| *deliver_in_udp_1* | *deliver_in_udp_2* | *deliver_in_udp_3*
| *deliver_in_99* | *deliver_in_99a*
| *timer_tt_rexmt_1*
| *timer_tt_rexmtsyn_1*
| *timer_tt_persist_1*
| *timer_tt_2msl_1*
| *timer_tt_delack_1*
| *timer_tt_conn_est_1*
| *timer_tt_keep_1*
| *timer_tt_fin_wait_2_1*
| *deliver_out_1*
| *deliver_out_99*
| *deliver_loop_99*
| *trace_1* | *trace_2*
| *interface_1*
| *epsilon_1*
| *epsilon_2*

Part IX

TCP1_timers

Chapter 9

Timers

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

1. the deadline timer ($'a$ timed), which wraps a value in a timer that will count towards a (possibly fuzzy) deadline, and stop the progress of time when it reaches the maximum deadline.
2. the time-window timer ($'a$ timewindow), which wraps a value in a timer just like a deadline timer, except that the value merely vanishes when it expires, rather than impeding the progress of time.

   These are an optimisation, designed to avoid having an extra rule (and consequent τ transitions) just for processing the expiry of such values.
3. the ticker (ticker), which contains a ts_seq (integral wraparound 32-bit type) that is incremented by one for every time a certain interval passes. It also contains the real remainder, and the interval size that corresponds to a step.
4. the stopwatch (stopwatch), which may be reset at any time and counts upwards indefinitely from zero. Note it may be necessary to add some fuzziness to this timer.

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

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

9.1 Properties (TCP and UDP)

Axioms of time, that all timers must satisfy.

9.1.1 Summary

time_pass_additive
time_pass_trajectory
opttorel

9.1.2 Rules

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

$$= \forall dur_1\ dur_2\ s_0\ s_1\ s_2.$$
$$\quad time\_pass\ dur_1\ s_0\ s_1 \wedge time\_pass\ dur_2\ s_1\ s_2 \implies time\_pass(dur_1 + dur_2) s_0\ s_2$$

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

– :

(time_pass_trajectory : (duration $\to {}'a \to {}'a \to$ bool) $\to$ bool)

$$
\begin{array}{l}
\qquad\qquad time\_pass \\
= \forall dur\ s_0\ s_1. \\
\quad time\_pass\ dur\ s_0\ s_1 \\
\quad\implies \\
\quad \exists w. \\
\qquad w\ 0 = s_0 \wedge \\
\qquad w\ dur = s_1 \wedge \\
\qquad \forall t\ t'. \\
\qquad\quad 0 \le t \wedge t \le dur \wedge \\
\qquad\quad 0 \le t' \wedge t' \le dur \wedge \\
\qquad\quad t < t' \\
\qquad\quad \implies \\
\qquad\quad time\_pass(t' - t)(w\ t)(w\ t')
\end{array}
$$

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

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

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

– :

(opttorel : (duration $\to {}'a \to {}'a$ option) $\to$ (duration $\to {}'a \to {}'a \to$ bool))

$$
\begin{array}{l}
\qquad tp\ dur\ x\ y \\
= \textbf{case}\ \ tp\ dur\ x\ \ \textbf{of} \\
\qquad \uparrow x' \to y = x' \\
\quad \| * \to \mathbf{F}
\end{array}
$$

Description Impedance-matching coercion.

9.2 Basic timer timer (TCP and UDP)

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

9.2.1 Summary

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

9.2.2 Rules

– :
timer = TIMER **of** duration #time#time

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

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

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

– timer that never goes off :
never_timer = TIMER$(0, \infty, \infty)$

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

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

– state of timer after time passage :
(Time_Pass_timer : duration → timer → timer option)
dur(TIMER$(e, deadmin, deadmax)$)
= **let** $e' = e + dur$
in
if time $e' \leq deadmax$
then $\uparrow$(TIMER$(e', deadmin, deadmax)$)
else $*$

9.3 Deadline timer timed (TCP and UDP)

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

9.3.1 Summary

timed
timed_val_of
timed_timer_of
timed_expires
Time_Pass_timed

9.3.2 Rules

– :
timed = TIMED **of** $'a\#$timer

– :
timed_val_of$((x)_{_}) = x$

– :
timed_timer_of$((x)_d) = d$

– :
timed_expires$((_)_d) =$ timer_expires d

– :
(Time_Pass_timed : duration $\rightarrow$ $'a$ timed $\rightarrow$ $'a$ timed option)
$dur((x)_d)$
$=$ **case** Time_Pass_timer dur d **of**
$\uparrow d' \rightarrow \uparrow((x)_{d'})$
$\| * \rightarrow *$

9.4 Time-window timer timewindow (TCP and UDP)

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

9.4.1 Summary

timewindow
timewindow_val_of
timewindow_open
Time_Pass_timewindow

9.4.2 Rules

– :
$\mathsf{timewindow} = \textsc{TimeWindow}\ \mathbf{of}\ {}'a\#\mathsf{timer} \mid \textsc{TimeWindowClosed}$

– :
$\mathrm{timewindow\_val\_of}((x)^{\textsc{TimeWindow}}_{\_}) = \uparrow x\ \wedge$
$\mathrm{timewindow\_val\_of}\ \ \textsc{TimeWindowClosed} = *$

– :
$\mathrm{timewindow\_open}((\_)^{\textsc{TimeWindow}}_{\_}) = \mathbf{T}\ \wedge$
$\mathrm{timewindow\_open}\ \ \textsc{TimeWindowClosed} = \mathbf{F}$

– :
$(\mathrm{Time\_Pass\_timewindow} : \mathrm{duration} \rightarrow {}'a\ \mathsf{timewindow} \rightarrow {}'a\ \mathsf{timewindow} \rightarrow \mathsf{bool})$
$dur((x)^{\textsc{TimeWindow}}_{d})tw'$
$= (\mathbf{case}\ \ \mathrm{Time\_Pass\_timer}\ \ dur\ d\ \ \mathbf{of}$
$\quad * \rightarrow tw' = \textsc{TimeWindowClosed}$
$\parallel \uparrow d' \rightarrow tw' = (x)^{\textsc{TimeWindow}}_{d'} \vee$
$\qquad (\mathrm{timer\_expires}\ \ d' \wedge tw' = \textsc{TimeWindowClosed})) \wedge$
$\mathrm{Time\_Pass\_timewindow}\ \ dur\ \textsc{TimeWindowClosed}\ tw' = (tw' = \textsc{TimeWindowClosed})$

9.5 Ticker ticker (TCP and UDP)

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

9.5.1 Summary

ticker
ticks_of
Time_Pass_ticker
ticker_ok
tick_imin
tick_imax

9.5.2 Rules

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

– **:**
$\text{ticks\_of}(\text{TICKER}(ticks, \_, \_, \_)) = ticks$

– **:**
$$
\begin{aligned}
&(\text{Time\_Pass\_ticker} : \text{duration} \rightarrow \mathsf{ticker} \rightarrow \mathsf{ticker} \rightarrow \mathsf{bool})\\
&dur(\text{TICKER}(ticks, remdr, intvlmin, intvlmax))t'\\
&\ = \textbf{let}\ \ d = remdr + dur\\
&\ \textbf{in}\\
&\quad \exists delta\ remdr'.\\
&\qquad d - \textbf{real\_of\_num}\ \ delta * intvlmax \leq remdr' \wedge\\
&\qquad remdr' \leq d - \textbf{real\_of\_num}\ \ delta * intvlmin \wedge\\
&\qquad 0 \leq remdr' \wedge remdr' < intvlmax \wedge\\
&\qquad t' = \text{TICKER}(ticks + delta, remdr', intvlmin, intvlmax)
\end{aligned}
$$

– **:**
$\text{ticker\_ok}(\text{TICKER}(ticks, remdr, imin, imax)) =$
$(0 \leq remdr \wedge remdr < imax \wedge imin \leq imax \wedge 0 < imin)$

– **:**
$\text{tick\_imin}(\text{TICKER}(t, r, imin, imax)) = imin$

– **:**
$\text{tick\_imax}(\text{TICKER}(t, r, imin, imax)) = imax$

9.6 Stopwatch stopwatch (TCP and UDP)

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

9.6.1 Summary

stopwatch
stopwatch_val_of
Time_Pass_stopwatch

9.6.2 Rules

– :
$\mathsf{stopwatch} = \textsc{Stopwatch}$ **of** duration (* may be zero *)$\#real\#real$

– :
$\mathrm{stopwatch\_val\_of}(\textsc{Stopwatch}(d, \_, \_)) = d$

– :
$(\mathrm{Time\_Pass\_stopwatch} : \mathrm{duration} \rightarrow \mathsf{stopwatch} \rightarrow \mathsf{stopwatch} \rightarrow \mathsf{bool})$
$dur(\textsc{Stopwatch}(d, ratemin, ratemax))s'$
$= \exists rate.ratemin \leq rate \wedge rate \leq ratemax \wedge$
$s' = \textsc{Stopwatch}(d + (dur * rate), ratemin, ratemax)$

Part X

TCP1_hostTypes

Chapter 10

Host types

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

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

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

10.1 Files (TCP and UDP)

10.1.1 Summary

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

10.1.2 Rules

– **file ID :**
$fid = \mathrm{FID}$ **of** num

– **socket ID :**
sid = SID **of** num

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

– **type of file, with pointer to details structure :**
filetype = FT_CONSOLE | FT_SOCKET **of** sid

– **flags set on a file :**
fileflags = ⟨[b : filebflag → bool]⟩

– **open file description :**

file $=\langle\!|$ *ft* : filetype; *ff* : fileflags$|\!\rangle$

– **helper constructor :**
FILE(*ft*, *ff*) $=\langle\!|$ *ft* := *ft*; *ff* := *ff* $|\!\rangle$

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

10.2 TCP states (TCP only)

10.2.1 Summary

tcpstate	TCP protocol states

10.2.2 Rules

– **TCP protocol states :**

```
tcpstate = CLOSED
         | LISTEN
         | SYN_SENT
         | SYN_RECEIVED
         | ESTABLISHED
         | CLOSE_WAIT
         | FIN_WAIT_1
         | CLOSING
         | LAST_ACK
         | FIN_WAIT_2
         | TIME_WAIT
```

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

10.3 The TCP control block (TCP only)

10.3.1 Summary

tcpReassSegment	segment reassembly queue elements
rexmtmode	retransmission mode
rttinf	round-trip time calculation parameters
tcpcb	the TCP control block

10.3.2 Rules

– **segment reassembly queue elements :**

tcpReassSegment
$=\langle\!|$ *seq* : tcp_seq_foreign;
spliced_urp : tcp_seq_foreign option;
FIN : bool;
data : *byte* list
$|\!\rangle$

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

– **retransmission mode :**
rexmtmode =
RexmtSyn
| Rexmt
| Persist

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

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

Description
This collects data used for round-trip time estimation.

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

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

– the TCP control block :

tcpcb =⟦

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

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

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

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

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

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

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

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

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

10.4 Sockets (TCP and UDP)

10.4.1 Summary

iobc	out-of-band data and status
socket_listen	extra info for a listening socket
tcp_socket	details of a TCP socket
dgram_msg	ordinary datagram on UDP receive queue
dgram_error	error (pseudo-)datagram on UDP receive queue
dgram	receive queue elements for a UDP socket
udp_socket	details of a UDP socket
sockflags	flags set on a 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

10.4.2 Rules

– **out-of-band data and status :**
iobc = NO_OOBDATA
 | OOBDATA **of** *byte*
 | HAD_OOBDATA

– **extra info for a listening socket :**

socket_listen
$=\langle\!|$ q_0 : sid list; (* incomplete connections queue *)
q : sid list; (* completed connections queue *)
$qlimit$: int(* backlog value as passed to listen *)
$|\!\rangle$

– **details of a TCP socket :**
tcp_socket
$=\langle\!|$ st : tcpstate; (* here rather than in tcpcb for convenience as heavily used. Called `t_state` in BSD *)
cb : tcpcb;
lis : socket_listen option; (* invariant: $*$ iff not LISTEN *)
$sndq$: $byte$ list;
$sndurp$: num option;
$rcvq$: $byte$ list;
$rcvurp$: num option; (* was "oobmark" *)
$iobc$: $iobc$
$|\!\rangle$

– **ordinary datagram on UDP receive queue :**
dgram_msg
$=\langle\!|$ $data$: $byte$ list;
is : ip option; (* source ip *)
ps : port option(* source port *)
$|\!\rangle$

– **error (pseudo-)datagram on UDP receive queue :**
dgram_error
$=\langle\!|$ e : error$|\!\rangle$

– **receive queue elements for a UDP socket :**
dgram = DGRAM_MSG **of** dgram_msg
| DGRAM_ERROR **of** dgram_error

– **details of a UDP socket :**
udp_socket
$=\langle\!|$ $rcvq$: dgram list$|\!\rangle$

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

Variations

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

– **flags set on a socket :**
sockflags = $\langle\!|$ b : sockbflag → bool;
n : socknflag → num;
t : socktflag → time
$|\!\rangle$

– **protocol-specific socket data :**
protocol_info = TCP_PROTO **of** tcp_socket
| UDP_PROTO **of** udp_socket

– **details of a socket :**
socket
= $\langle\!|$ *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 *)
$|\!\rangle$

– **helper constructor :**
TCP_Sock0(*st*, *cb*, *lis*, *sndq*, *sndurp*, *rcvq*, *rcvurp*, *iobc*)
= $\langle\!|$ *st* := *st*; *cb* := *cb*; *lis* := *lis*; *sndq* := *sndq*;
sndurp := *sndurp*; *rcvq* := *rcvq*; *rcvurp* := *rcvurp*; *iobc* := *iobc* $|\!\rangle$

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

– **helper constructor :**
UDP_Sock0(*rcvq*) = $\langle\!|$ *rcvq* := *rcvq* $|\!\rangle$

– **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*)
= $\langle\!|$ *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* $|\!\rangle$

– **helper accessor (beware ARBitrary behaviour on non-TCP socket) :**
tcp_sock_of *sock* = **case** *sock.pr* **of** TCP_PROTO(*tcp_sock*) → *tcp_sock* ‖ _ → *ARB*

– **helper accessor (beware ARBitrary behaviour on non-UDP socket) :**
udp_sock_of *sock* = **case** *sock.pr* **of** UDP_PROTO(*udp_sock*) → *udp_sock* ‖ _ → *ARB*

– **helper accessor :**
proto_of(TCP_PROTO(*_1*)) = PROTO_TCP ∧
proto_of(UDP_PROTO(*_3*)) = PROTO_UDP

– **compare protocol of two protocol info structures :**
proto_eq *pr* pr' = (proto_of *pr* = proto_of pr')

Description Various convenience functions.

10.5 The host (TCP and UDP)

10.5.1 Summary

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

10.5.2 Rules

– **the architectures we consider :**

arch = LINUX_2_4_20_8
  | WINXP_PROF_SP1
  | FREEBSD_4_6_RELEASE

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

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

– **network interface descriptor :**

ifd =⟨[*ipset* : ip set; (* set of IP addresses of this interface *)
  primary : ip; (* and the primary IP address *)
  netmask : *netmask*; (* netmask *)
  up : bool(* status: up (and connected) or not *)
]⟩

– **routing table entry :**

routing_table_entry =⟨[*destination_ip* : ip;
  destination_netmask : *netmask*;
  ifid : *ifid*
]⟩

Description

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

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

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

Rule version: $Id: TCP1_hostTypesScript.sml,v 1.155 2005/03/16 15:06:36 pes20 Exp $

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

– :
type_abbrev *routing_table* : routing_table_entry list

– segment category, determining which band limiter to use :
bandlim_reason = BANDLIM_UNLIMITED
| BANDLIM_RST_CLOSEDPORT
| BANDLIM_RST_OPENPORT

Description internal bandlimiter state; intended to be opaque

– :
type_abbrev *bandlim_state* : (tcpSegment# ts_seq #bandlim_reason)list

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

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

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

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

10.6 Trace records (TCP and UDP)

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

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

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

10.6.1 Summary

traceflavour	trace record flavours
type_abbrev_tracerecord	
tracecb_eq	compare two control blocks for ”equality” modulo known issues
tracesock_eq	compare two sockets for ”equality” modulo known issues

10.6.2 Rules

– **trace record flavours :**

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

Description Different situations in which a trace may be generated.

– **:**

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

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

tracecb_eq(flav : traceflavour)(st : tcpstate)(es : error option)(cb : tcpcb)(cb' : tcpcb)
$= ((cb.snd\_una = cb'.snd\_una) \wedge$
(**if** flav = TA_OUTPUT **then T else** $cb.snd\_max = cb'.snd\_max) \wedge$
(**if** flav = TA_OUTPUT $\vee$ (st = SYN_SENT $\wedge$ $es \neq *$)
then T
else $cb.snd\_nxt = cb'.snd\_nxt) \wedge$ (* only bad on error *)
$(cb.snd\_wl1 = cb'.snd\_wl1) \wedge$
$(cb.snd\_wl2 = cb'.snd\_wl2) \wedge$
$(cb.iss = cb'.iss) \wedge$
$(cb.snd\_wnd = cb'.snd\_wnd) \wedge$
(**if** flav = TA_OUTPUT **then T else** $cb.snd\_cwnd = cb'.snd\_cwnd) \wedge$ (* only bad on error *)
$(cb.snd\_ssthresh = cb'.snd\_ssthresh) \wedge$

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

(* (**if** st = SYN_SENT **then T else** $cb.rcv\_wnd = cb'.rcv\_wnd)\wedge$ *)

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

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

$(cb.rcv\_nxt = cb'.rcv\_nxt) \wedge$
$(cb.rcv\_up = cb'.rcv\_up) \wedge$
$(cb.irs = cb'.irs) \wedge$
(**if** flav = TA_OUTPUT $\vee$ flav = TA_INPUT **then T else** $cb.rcv\_adv = cb'.rcv\_adv) \wedge$
(**if** flav = TA_OUTPUT $\vee$ st = SYN_SENT $\vee$ st = TIME_WAIT
(* we store our initially-sent MSS in *t_maxseg*, whereas BSD just recalculates it. This test decouples the model from BSD in order to cope with this. *)
then T else $cb.t\_maxseg = cb'.t\_maxseg) \wedge$ (* only bad on error *)
$(cb.t\_dupacks = cb'.t\_dupacks) \wedge$
$(cb.snd\_scale = cb'.snd\_scale) \wedge$
$(cb.rcv\_scale = cb'.rcv\_scale) \wedge$
(* t_rtseq, if t_rtttime <> 0; ignore t_rtttime *)(* only bad on error *)
(**if** flav = TA_OUTPUT $\vee$ flav = TA_INPUT **then T else**
option_map snd $cb.t\_rttseg$ = **option_map snd** $cb'.t\_rttseg) \wedge$
(timewindow_val_of $cb.ts\_recent$ = timewindow_val_of $cb'.ts\_recent) \wedge$
(**if** flav = TA_OUTPUT $\vee$ flav = TA_INPUT **then T else** $cb.last\_ack\_sent = cb'.last\_ack\_sent$))
(* also ignore, always: *tt_delack*; in case of error: *tt_rexmt*, *t_softerror* *)

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

tracesock_eq(flav, sid, $quad$, st, cb)sid' $sock$
= (proto_of $sock.pr$ = PROTO_TCP $\wedge$
let tcp_sock = tcp_sock_of $sock$ **in**
sid $= sid' \wedge$
(* 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) *)
(**case** $quad$ **of**
$\uparrow(is_1, ps_1, is_2, ps_2) \to is_1 = sock.is_1 \wedge$
$ps_1 = sock.ps_1 \wedge$
(**if** flav = TA_DROP **then T else** $is_2 = sock.is_2) \wedge$
(**if** flav = TA_DROP **then T else** $ps_2 = sock.ps_2) \parallel$
$* \to$ **T**) $\wedge$
$st = tcp\_sock.st \wedge$

tracecb_eq *flav st sock.es cb tcp_sock.cb*)

Part XI

TCP1_params

Chapter 11

Host behavioural parameters

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

11.1 Model parameters (TCP and UDP)

Booleans that select a particular model semantics.

11.1.1 Summary

INFINITE_RESOURCES
BSD_RTTVAR_BUG

11.1.2 Rules

– :
INFINITE_RESOURCES = **T**

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

– :
BSD_RTTVAR_BUG = **T**

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

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

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

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

11.2 Scheduling parameters (TCP and UDP)

Parameters controlling the timing of the OS scheduler.

11.2.1 Summary

dschedmax
diqmax
doqmax

11.2.2 Rules

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

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

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

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

11.3 Timers (TCP and UDP)

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

11.3.1 Summary

HZ
tickintvlmin
tickintvlmax
stopwatchfuzz
stopwatch_zero
SLOW_TIMER_INTVL
SLOW_TIMER_MODEL_INTVL
FAST_TIMER_INTVL
FAST_TIMER_MODEL_INTVL
KERN_TIMER_INTVL
KERN_TIMER_MODEL_INTVL

11.3.2 Rules

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

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

– :
tickintvlmin = $100/(105 * \mathrm{HZ})$: *real*

– :
tickintvlmax = $105/(100 * \mathrm{HZ})$: *real*

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

– :
stopwatchfuzz = (5/100) : *real*(* +/- factor on accuracy of stopwatch timers *)

– :
stopwatch_zero = $\textsc{Stopwatch}(0, 1/(1 + \text{stopwatchfuzz}), 1 + \text{stopwatchfuzz})$

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

– :
SLOW_TIMER_INTVL = (1/2) : duration (* slow timer is 500msec on BSD *)

– :
SLOW_TIMER_MODEL_INTVL = (1/1000) : duration (* 1msec fuzziness to mask atomicity of model; Note that it might be possible to reduce this fuzziness *)

– :
FAST_TIMER_INTVL = (1/5) : duration (* fast timer is 200msec on BSD *)

– :
FAST_TIMER_MODEL_INTVL = (1/1000) : duration (* 1msec fuzziness to mask atomicity of model; Note that it might be possible to reduce this fuzziness *)

– :
KERN_TIMER_INTVL = tickintvlmax : duration (* precision of select timer *)

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

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

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

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

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

11.4.1 Summary

privileged_ports
ephemeral_ports
OPEN_MAX
OPEN_MAX_FD
FD_SETSIZE
SOMAXCONN

11.4.2 Rules

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

– :
$\text{ephemeral\_ports} = \{\textsc{Port}\ n \mid n \geq 1024 \land n \leq 5000\}$

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

– :
OPEN_MAX = 957 : num (* typical value of `kern.maxfilesperproc` on one of our BSD boxen *)

– :
OPEN_MAX_FD = FD OPEN_MAX

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

– :
$(\text{FD\_SETSIZE} : \mathit{arch} \rightarrow \mathsf{num})\textsc{Linux\_2\_4\_20\_8} = 1024n \land$
$\text{FD\_SETSIZE}\ \ \textsc{WinXP\_Prof\_SP1} = 64n \land$
$\text{FD\_SETSIZE}\ \ \mathit{FreeBDS\_4\_6\_RELEASE} = 1024n$

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

Variations

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

– :
SOMAXCONN = 128 : num

Description The maximum listen-queue length.

11.5 UDP parameters (UDP only)

UDP-specific parameters.

11.5.1 Summary

UDPpayloadMax

11.5.2 Rules

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

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

11.6 Buffers (TCP and UDP)

Parameters to the buffer size computation.

11.6.1 Summary

MCLBYTES	size of an mbuf cluster
MSIZE	
SB_MAX	
oob_extra_sndbuf	

11.6.2 Rules

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

– :
MSIZE = 256 : num(* BSD default on i386; really, size of an mbuf *)

– :
SB_MAX = 256 ∗ 1024 : num(* BSD *)

– :
oob_extra_sndbuf = 1024 : num

11.7 File and socket flag defaults (TCP and UDP)

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

11.7.1 Summary

ff_default_b	file flags default
ff_default	
sf_default_b	bool socket flags default
sf_default_n	num socket flags defaults
sf_default_t	time socket flags defaults
sf_default	socket flags defaults
sf_min_n	minimum values of num socket flags
sf_max_n	maximum values of num socket flags
sndrcv_timeo_t_max	maximum value of send/recv timeouts
pselect_timeo_t_max	maximum value of pselect timeouts

11.7.2 Rules

– file flags default :
(ff_default_b : filebflag → bool)
O_NONBLOCK = **F** ∧
ff_default_b O_ASYNC = **F**

– :
ff_default =⟨| b := ff_default_b|⟩

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

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

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

sf_default_n FreeBSD_4_6_RELEASE SOCK_STREAM SO_RCVBUF = 57344 ∧ (* from code *)

sf_default_n Linux_2_4_20_8 SOCK_STREAM SO_SNDLOWAT = 1 ∧ (* from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_STREAM SO_SNDLOWAT = 1 ∧ (* Note this value has not been checked in testi
sf_default_n FreeBSD_4_6_RELEASE SOCK_STREAM SO_SNDLOWAT = 2048 ∧ (* from code *)

sf_default_n Linux_2_4_20_8 SOCK_STREAM SO_RCVLOWAT = 1 ∧ (* from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_STREAM SO_RCVLOWAT = 1 ∧
sf_default_n FreeBSD_4_6_RELEASE SOCK_STREAM SO_RCVLOWAT = 1 ∧ (* from code *)

sf_default_n Linux_2_4_20_8 SOCK_DGRAM SO_SNDBUF = 65535 ∧ (* from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_DGRAM SO_SNDBUF = 8192 ∧ (* from tests *)
sf_default_n FreeBSD_4_6_RELEASE SOCK_DGRAM SO_SNDBUF = 9216 ∧ (* from code *)

sf_default_n Linux_2_4_20_8 SOCK_DGRAM SO_RCVBUF = 65535 ∧ (* correct from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_DGRAM SO_RCVBUF = 8192 ∧ (* correct from tests *)
sf_default_n FreeBSD_4_6_RELEASE SOCK_DGRAM SO_RCVBUF = 42080∧ (* from tests but: 41600 from code; i386 only as dependent on `sizeof(struct sockaddr_in)` *)

sf_default_n Linux_2_4_20_8 SOCK_DGRAM SO_SNDLOWAT = 1 ∧ (* from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_DGRAM SO_SNDLOWAT = 1 ∧ (* from tests *)
sf_default_n FreeBSD_4_6_RELEASE SOCK_DGRAM SO_SNDLOWAT = 2048 ∧ (* from code *)

sf_default_n Linux_2_4_20_8 SOCK_DGRAM SO_RCVLOWAT = 1 ∧ (* from tests *)
sf_default_n WinXP_Prof_SP1 SOCK_DGRAM SO_RCVLOWAT = 1 ∧ (* from tests *)
sf_default_n FreeBSD_4_6_RELEASE SOCK_DGRAM SO_RCVLOWAT = 1(* from code *)

– time **socket flags defaults :**
(sf_default_t : socktflag → time)
SO_LINGER = ∞ ∧
sf_default_t SO_SNDTIMEO = ∞ ∧
sf_default_t SO_RCVTIMEO = ∞

– **socket flags defaults :**
sf_default *arch* socktype =⦇ *b* := sf_default_b;
n := sf_default_n *arch* socktype;
t := sf_default_t
⦈

– **minimum values of** num **socket flags :**
(sf_min_n : *arch* → socknflag → num)
Linux_2_4_20_8 SO_SNDBUF = 2048 ∧ (* from tests *)
sf_min_n WinXP_Prof_SP1 SO_SNDBUF = 0 ∧ (* from tests *)
sf_min_n FreeBSD_4_6_RELEASE SO_SNDBUF = 1 ∧ (* from code *)
sf_min_n Linux_2_4_20_8 SO_RCVBUF = 256 ∧ (* from tests *)
sf_min_n WinXP_Prof_SP1 SO_RCVBUF = 0 ∧ (* from tests *)
sf_min_n FreeBSD_4_6_RELEASE SO_RCVBUF = 1 ∧ (* from code *)
sf_min_n Linux_2_4_20_8 SO_SNDLOWAT = 1 ∧ (* from tests *)

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

– **maximum values of** num **socket flags :**
(sf_max_n : *arch* → socknflag → num)
LINUX_2_4_20_8 SO_SNDBUF = 131070 ∧ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_SNDBUF = 131070 ∧ (* from tests *)
sf_max_n FREEBSD_4_6_RELEASE SO_SNDBUF =
SB_MAX ∗ MCLBYTES **div**(MCLBYTES + MSIZE) ∧ (* from code *)
sf_max_n LINUX_2_4_20_8 SO_RCVBUF = 131070 ∧ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_RCVBUF = 131070 ∧ (* from tests *)
sf_max_n FREEBSD_4_6_RELEASE SO_RCVBUF =
SB_MAX ∗ MCLBYTES **div**(MCLBYTES + MSIZE) ∧ (* from code *)
sf_max_n LINUX_2_4_20_8 SO_SNDLOWAT = 1 ∧ (* from tests *)
sf_max_n WINXP_PROF_SP1 SO_SNDLOWAT = 1 ∧ (* Note this value has not been checked in testing. *)
sf_max_n FREEBSD_4_6_RELEASE SO_SNDLOWAT =
SB_MAX ∗ MCLBYTES **div**(MCLBYTES + MSIZE) ∧ (* clip to SO_SNDBUF *)
sf_max_n LINUX_2_4_20_8 SO_RCVLOWAT = **w2n** *INT32_SIGNED_MAX* ∧ (* from code *)
sf_max_n WINXP_PROF_SP1 SO_RCVLOWAT = 1 ∧ (* Note this value has not been checked in testing. *)
sf_max_n FREEBSD_4_6_RELEASE SO_RCVLOWAT =
SB_MAX ∗ MCLBYTES **div**(MCLBYTES + MSIZE)(* clip to SO_RCVBUF *)

– **maximum value of** send/recv **timeouts :**
sndrcv_timeo_t_max = time 655350000

– **maximum value of** pselect **timeouts :**
pselect_timeo_t_max = time(31 ∗ 24 ∗ 3600)

11.8 RFC-specified limits (TCP only)

Protocol value limits specified in the TCP RFCs.

11.8.1 Summary

dtsinval	RFC1323 s4.2.3: timestamp validity period.
TCP_MAXWIN	maximum (scaled) window size
TCP_MAXWINSCALE	maximum window scaling exponent

11.8.2 Rules

– **RFC1323 s4.2.3: timestamp validity period. :**
dtsinval = time(24 ∗ 24 ∗ 60 ∗ 60)

– **maximum (scaled) window size :**
TCP_MAXWIN = 65535 : num

– **maximum window scaling exponent :**
TCP_MAXWINSCALE = 14 : num

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

11.9 Protocol parameters (TCP only)

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

11.9.1 Summary

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

11.9.2 Rules

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

– **initial** *snd_cwnd* **for local connections :**
SS_FLTSZ_LOCAL = 4 : num(* BSD; is a sysctl *)

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

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

– **:**
TCP_Q0MINLIMIT = 30 : num(* FreeBSD 4.6-RELEASE: tcp_syncache.bucket_limit *)

– **:**
TCP_Q0MAXLIMIT = $512 * 30$: num(* FreeBSD 4.6-RELEASE: tcp_syncache.cache_limit *)

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

– :
backlog_fudge(n : int) = **min** SOMAXCONN(clip_int_to_num n)

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

11.10 Time values (TCP only)

Various time intervals controlling TCP's behaviour.

11.10.1 Summary

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

11.10.2 Rules

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

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

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

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

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

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

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

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

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

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

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

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

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

11.11 Timing-related parameters (TCP only)

Parameters relating to TCP's exponential backoff.

11.11.1 Summary

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

11.11.2 Rules

– **TCP exponential retransmit backoff: BSD: from source code, tcp_timer.c:155 :**
TCP_BSD_BACKOFFS = [1; 2; 4; 8; 16; 32; 64; 64; 64; 64; 64; 64; 64] : num list

– **TCP exponential retransmit backoff: Linux: experimentally determined :**
TCP_LINUX_BACKOFFS = [1; 2; 4; 8; 16; 32; 64; 128; 256; 512; 512] : num list(* Note: the tail may be incomplete *)

– **TCP exponential retransmit backoff: WinXP: experimentally determined :**
TCP_WINXP_BACKOFFS = [1; 2; 4; 8; 16] : num list(* Note: the tail may be incomplete *)

– **TCP maximum retransmit shift :**
TCP_MAXRXTSHIFT = 12 : num(* TCPv2p842 *)

– **TCP maximum SYNACK retransmit shift :**
TCP_SYNACKMAXRXTSHIFT = 3 : num(* FreeBSD 4.6-RELEASE, tcp_syncache.c:SYNCACHE_MAXREXMTS *)

– **TCP exponential SYN retransmit backoff: BSD: tcp_timer.c:152 :**
TCP_SYN_BSD_BACKOFFS = [1; 1; 1; 1; 1; 2; 4; 8; 16; 32; 64; 64; 64] : num list(* Our experimentation shows that this list stops at 8. This will be due to the connection establishment timer firing. Values here are obtained from the BSD source *)

– **TCP exponential SYN retransmit backoff: Linux: experimentally determined :**
TCP_SYN_LINUX_BACKOFFS = [1; 2; 4; 8; 16] : num list(* This list might be longer. Experimentation does not show further entries, perhaps due to the connection establishment timer firing *)

– **TCP exponential SYN retransmit backoff: WinXP: experimentally determined :**
TCP_SYN_WINXP_BACKOFFS = [1; 2] : num list(* This list might be longer. Experimentation does not show further entries, perhaps due to the connection establishment timer firing *)

Part XII

TCP1_auxFns

Chapter 12

Auxiliary functions

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

12.1 Architecture handling (TCP and UDP)

Many aspects of host behaviour differ from one OS to another, and so a host has an architecture parameter detailing its precise OS and version (e.g., LINUX_2_4_20_8). Very often, however, we do not need to be so precise – a certain behaviour might apply to all Linux, or even all Unix, OSes. Below we define predicates for these cases, to allow variant architectures to be easily added later.

12.1.1 Summary

windows_arch	test if host architecture is Windows
bsd_arch	test if host architecture is BSD
linux_arch	test if host architecture is Linux
unix_arch	test if host architecture is Unix

12.1.2 Rules

– **test if host architecture is Windows :**
windows_arch $arch = (arch \in \{\textsc{WinXP\_Prof\_SP1}\})$

– **test if host architecture is BSD :**
bsd_arch $arch = (arch \in \{\textsc{FreeBSD\_4\_6\_RELEASE}\})$

– **test if host architecture is Linux :**
linux_arch $arch = (arch \in \{\textsc{Linux\_2\_4\_20\_8}\})$

– **test if host architecture is Unix :**
unix_arch $arch = (arch \in \{\textsc{Linux\_2\_4\_20\_8}; \textsc{FreeBSD\_4\_6\_RELEASE}\})$

12.2 Interfaces and IP addresses (TCP and UDP)

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

12.2.1 Summary

mask	apply a netmask to an IP to obtain the network number
mask_bits	compute network bitmask from netmask

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

12.2.2 Rules

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

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

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

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

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

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

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

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

– **is IP address a loopback address? :**
$\text{in\_loopback}\ i = (i \in \text{LOOPBACK\_ADDRS})$

– **is IP address a local address? :**
$\text{in\_local}(\mathit{ifds} : \mathit{ifid} \mapsto \mathsf{ifd})i =$
$\quad(\text{in\_loopback}\ i \lor$
$\quad i \in (\mathbf{bigunion}\{\mathit{ifd\_.ipset} \mid \mathit{ifd\_} \in (\mathbf{rng}(\mathit{ifds}))\}))$

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

– **the set of local IP addresses :**
$\text{local\_ips}(\mathit{ifds} : \mathit{ifid} \mapsto \mathsf{ifd}) = \mathbf{bigunion}\{\mathit{ifd\_.ipset} \mid \mathit{ifd\_} \in (\mathbf{rng}(\mathit{ifds}))\}$
(* annoying: ifd is a constructor, and { | } has no binder to allow us to shadow it *)

– **the set of local primary IP addresses :**
$\mathit{local\_primary\_ips}(\mathit{ifds} : \mathit{ifid} \mapsto \mathsf{ifd}) = \{\mathit{ifd\_.primary} \mid \mathit{ifd\_} \in (\mathbf{rng}(\mathit{ifds}))\}$

– **is IP address on a local subnet of this host? :**
$\text{is\_localnet}(\mathit{ifds}_0 : \mathit{ifid} \mapsto \mathit{ifd})i =$
$(\exists \mathit{ifd}.\mathit{ifd} \in (\mathbf{rng}(\mathit{ifds}_0)) \wedge \text{mask } \mathit{ifd.netmask}\ i = \text{mask } \mathit{ifd.netmask}\ \mathit{ifd.primary})$

– **is IP address a broadcast address? :**
$\text{if\_broadcast}(\mathit{ifd0} : \mathsf{ifd})$
$= \mathbf{case}\ (\mathit{ifd0.netmask}, \text{mask } \mathit{ifd0.netmask}\ \mathit{ifd0.primary})\ \mathbf{of}$
$\quad (\text{NETMASK } m, \mathsf{ip}\ n$(* n has been masked by m above *)$) \rightarrow$
$\quad\quad \mathsf{ip}(n + 2\ {**}\ (32 - m) - 1)$
(* Note: would be much easier if IPs were actually *word32* rather than num *)
(* corresponds to INADDR_BROADCAST for the interface *)

– **the set of addresses in an interface's subnet :**
$\text{if\_any}(\mathit{ifd0} : \mathsf{ifd})$
$= \mathbf{case}\ (\mathit{ifd0.netmask}, \text{mask } \mathit{ifd0.netmask}\ \mathit{ifd0.primary})\ \mathbf{of}$
$\quad (\text{NETMASK } m, \mathsf{ip}\ n$(* n has been masked by m above *)$) \rightarrow$
$\quad\quad \mathsf{ip}(n)$
(* Note: would be much easier if IPs were actually *word32* rather than num *)

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

– **is IP address a broadcast/multicast address? :**
$\text{is\_broadormulticast}(\mathit{ifds}_0 : \mathit{ifid} \mapsto \mathsf{ifd})i =$
$(i \in \text{IN\_MULTICAST} \vee$ (* is i a multicast address? *)
$i = \text{INADDR\_BROADCAST} \vee$ (* is i the default broadcast address? [CORRECT NAME?] *)
$\exists (k, \mathit{ifd0}) :: \mathit{ifds}_0.$
$\quad i \in \{\text{if\_broadcast } \mathit{ifd0};$ (* is i the broadcast addr for any interface? *)
$\quad\quad \text{if\_any } \mathit{ifd0}\})$ (* RFC 1122 - should accept an all-0s or all-1s broadcast address. all three OSes do *)

Description Test if IP address i is a broadcast or multicast address, wrt the given set of interfaces ifds_0. If no interfaces given ($\mathit{ifds}_0 = *$), then treat only INADDR_BROADCAST as a broadcast address.

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

– **compute set of routeable addresses for a routing table entry :**
$\text{routeable}(\mathit{rte} : \mathsf{routing\_table\_entry}) =$
$\{i \mid \text{mask } \mathit{rte.destination\_netmask}\ i = \text{mask } \mathit{rte.destination\_netmask}\ \mathit{rte.destination\_ip}\}$

– **determine list of possible sending interfaces :**
$\text{outroute\_ifids}(i_2, \mathit{rttab} : \mathit{routing\_table}) =$
$\text{MAP\_OPTIONAL}(\lambda \mathit{rte}.\mathbf{if}\ i_2 \in \text{routeable } \mathit{rte}\ \mathbf{then} \uparrow \mathit{rte.ifid}\ \mathbf{else} *)\mathit{rttab}$

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

– **is the interface up? :**
$\text{ifid\_up}\ \mathit{ifds}\ \mathit{ifid} = (\mathit{ifds}[\mathit{ifid}]).\mathit{up}$

– **compute interface to use to send to given IP, if any :**
$\text{outroute}(i_2, \mathit{rttab} : \mathit{routing\_table}, \mathit{ifds} : \mathit{ifid} \mapsto \mathsf{ifd}) =$
$\textbf{case}\ \ \textbf{filter}(\text{ifid\_up}\ \ \mathit{ifds})(\text{outroute\_ifids}(i_2, \mathit{rttab}))\ \textbf{of}$
$\quad [\,] \to *$
$\ \| \ (\mathit{ifid} :: \_\mathit{987}) \to \uparrow \mathit{ifid}$

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

– **compute source address to use to route to given IP :**
$\text{auto\_outroute}(\mathit{i2}', \uparrow i_2, \mathit{rttab}, \mathit{ifds}) = \{i_2\} \wedge$
$\text{auto\_outroute}(\mathit{i2}', *, \mathit{rttab}, \mathit{ifds}) = \textbf{case}\ \ \text{outroute}(\mathit{i2}', \mathit{rttab}, \mathit{ifds})\ \textbf{of}$
$\qquad \uparrow \mathit{ifid} \to \{(\mathit{ifds}[\mathit{ifid}]).\mathit{primary}\}$
$\qquad \| * \to \{\}$

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

– **test if we can route to given IP, returning appropriate error if not :**
$\text{test\_outroute\_ip}(i_2 : \mathsf{ip}, \mathit{rttab}, \mathit{ifds}, \mathit{arch})$
$= \textbf{let}\ \ \mathit{ifids} = \text{outroute\_ifids}(i_2, \mathit{rttab})\ \textbf{in}$
$\quad \textbf{if}\ \ \mathit{ifids} = [\,]\ \textbf{then}$
$\qquad (\textbf{if}\ \ \text{linux\_arch}\ \ \mathit{arch}\ \ \textbf{then}\ \uparrow \text{ENETUNREACH}$
$\qquad\ \textbf{else}\ \uparrow \text{EHOSTUNREACH})$
$\quad \textbf{else}$
$\qquad \textbf{if}\ \ \textbf{filter}(\text{ifid\_up}\ \ \mathit{ifds})\mathit{ifids} = [\,]\ \textbf{then}$
$\qquad\quad \uparrow \text{ENETDOWN}$
$\qquad \textbf{else}\ *$

– **if destination IP specified, do** test_outroute_ip **:**
$\text{test\_outroute}(\mathsf{msg} : \mathsf{msg}, \mathit{rttab}, \mathit{ifds}, \mathit{arch})$
$= \textbf{case}\ \ \mathsf{msg}.\mathit{is}_2\ \ \textbf{of}$
$\quad \uparrow i_2 \to \uparrow(\text{test\_outroute\_ip}(i_2, \mathit{rttab}, \mathit{ifds}, \mathit{arch}))$
$\ \| \ \_ \to *$

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

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

test_outroute modified to be functional rather than relational, as behaviour is purely deterministic. The result is of type error option option, where the first level of "optionality" indicates whether or not the function is even being called on valid input (whether or not message has an is_2 "field"), and the next level indicates errors being raised, or not.

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

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

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

Description RFC1122 says loopback addresses must never appear on the wire. Here we test if this segment is in violation. Ideally, we'd check "(src or dest in loopback net) and interface not loopback", but we can't see which interface it's going out of in this model. The condition above is possibly the best approximation we can make if one considers the possible values of $\mathsf{msg}.is_1$ and $\mathsf{msg}.is_2$.

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

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

12.3.1 Summary

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

12.3.2 Rules

– **< comparison on file descriptors :**

$\text{fdlt}(\text{FD}\ n)(\text{FD}\ m) = n < m$

– **≤ comparison on file descriptors :**

$\text{fdle}(\text{FD}\ n)(\text{FD}\ m) = n \leq m$

– **least fd satisfying predicate P :**

$\text{leastfd}\ P = \text{FD}(\textbf{least}\ n.P(\text{FD}\ n))$

– **next file descriptor to use :**

$\text{nextfd}\ \mathit{arch}\ \mathit{fds}\ \mathit{fd}' = \textbf{if}\ \text{windows\_arch}\ \mathit{arch}\ \textbf{then}$
(* no ordering on Windows fds; they're just handles *)
$\mathit{fd}' \notin \mathbf{dom}(\mathit{fds})$
$\textbf{else}$
(* POSIX architectures allocate in order *)
$\mathit{fd}' = \text{leastfd}\ \mathit{fd}'.\mathit{fd}' \notin \mathbf{dom}(\mathit{fds})$

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

Variations

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

– **count references to given** *fid* **:**
$\text{fid\_ref\_count}(\mathit{fds} : \mathsf{fd} \mapsto \mathit{fid}, \mathit{fid}) = \mathbf{card}(\mathbf{dom}((\mathbf{rrestrict}\ \mathit{fds}\{\mathit{fid}\})))$

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

– **socket sanity invariants hold :**
sane_socket *sock* = **case** *sock.pr* **of**
TCP_PROTO *tcp_sock* →
(*LENGTH tcp_sock.rcvq <= sock.sf.n(SO_RCVBUF) /\ (* true?? *)*)
length *tcp_sock.rcvq* ≤ TCP_MAXWIN ≪ TCP_MAXWINSCALE (*/*)
(*LENGTH tcp_sock.sndq <= sock.sf.n(SO_SNDBUF) (* true?? *)*)
‖ UDP_PROTO *udp_sock* →
T

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

12.4 Binding (TCP and UDP)

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

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

12.4.1 Summary

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

12.4.2 Rules

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

$\text{bound\_ports\_protocol\_autobind}\ \ pr\ socks = \{p \mid \exists s : \mathsf{socket}.$
$\quad s\ \in \mathbf{rng}(socks) \wedge s.ps_1 = \uparrow p \wedge$
$\quad \text{proto\_of}\ \ s.pr = pr\}$

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? :

$\text{bound\_port\_allowed}\ \ pr\ socks\ sf\ \ arch\ is\ p =$
$p\ \notin$
$\{\mathsf{port} \mid \exists s : \mathsf{socket}.$
$\quad s\ \in \mathbf{rng}(socks) \wedge s.ps_1 = \uparrow \mathsf{port} \wedge$
$\quad \text{proto\_eq}\ \ s.pr\ pr \wedge$
$\quad (\mathbf{if}\ \ \text{bsd\_arch}\ \ arch \wedge \text{SO\_REUSEADDR}\ \in\ sf.b\ \ \mathbf{then}$
$\qquad s.is_2 = * \wedge s.is_1 = is$
$\qquad \mathbf{else\ \ if}\ \ \text{linux\_arch}\ \ arch \wedge \text{SO\_REUSEADDR}\ \in\ sf.b \wedge \text{SO\_REUSEADDR}\ \in\ s.sf.b \wedge$
$\qquad\qquad ((\exists tcp\_sock.\text{TCP\_PROTO}(tcp\_sock) = s.pr \wedge \neg(tcp\_sock.st = \text{LISTEN})) \vee$
$\qquad\qquad\ \exists udp\_sock.\text{UDP\_PROTO}(udp\_sock) = s.pr)\ \mathbf{then}$
$\qquad \mathbf{F}$(* If socket is not in LISTEN state or is a UDP socket can always rebind here *)
$\qquad \mathbf{else\ \ if}\ \ \text{windows\_arch}\ \ arch \wedge \text{SO\_REUSEADDR}\ \in\ sf.b\ \ \mathbf{then}$
$\qquad \mathbf{F}$(* can rebind any UDP address; not sure about TCP - assume the same for now *)
$\qquad \mathbf{else}$
$\qquad (is = * \vee s.is_1 = * \vee (\exists i : ip.is = \uparrow i \wedge s.is_1 = \uparrow i)))\}$

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 :

$\text{autobind}(\uparrow p, \_, \_) = \{p\} \wedge$
$\text{autobind}(*, pr, socks) = \text{ephemeral\_ports}\ \ \mathbf{diff}(\text{bound\_ports\_protocol\_autobind}\ \ pr\ socks)$

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

– was sid bound more recently than sid'? :

$\text{bound\_after}\ \mathsf{sid}\ sid'[\,] = ASSERTION\_FAILURE$“bound_after”(* should never reach this case *) $\wedge$
$\text{bound\_after}\ \mathsf{sid}\ sid'(sid0 :: bound) =$
$\mathbf{if}\ \mathsf{sid} = sid0\ \ \mathbf{then}\ \ \mathbf{T}$(* newly-bound sockets are added to the head *)
$\mathbf{else\ \ if}\ \ sid' = sid0\ \ \mathbf{then}\ \ \mathbf{F}$
$\quad \mathbf{else}\ \ \text{bound\_after}\ \mathsf{sid}\ sid'\ bound$

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

$(\text{match\_score}(\_, *, \_, \_)\_ = 0n) \wedge$

$(\text{match\_score}(*, \uparrow p_1, *, *)(i_3, ps_3, i_4, ps_4) =$
$\quad \textbf{if } ps_4 = \uparrow p_1 \textbf{ then } 1 \textbf{ else } 0) \wedge$
$(\text{match\_score}(\uparrow i_1, \uparrow p_1, *, *)(i_3, ps_3, i_4, ps_4) =$
$\quad \textbf{if } (i_1 = i_4) \wedge (\uparrow p_1 = ps_4) \textbf{ then } 2 \textbf{ else } 0) \wedge$
$(\text{match\_score}(\uparrow i_1, \uparrow p_1, \uparrow i_2, *)(i_3, ps_3, i_4, ps_4) =$
$\quad \textbf{if } (i_2 = i_3) \wedge (i_1 = i_4) \wedge (\uparrow p_1 = ps_4) \textbf{ then } 3 \textbf{ else } 0) \wedge$
$(\text{match\_score}(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)(i_3, ps_3, i_4, ps_4) =$
$\quad \textbf{if } (\uparrow p_2 = ps_3) \wedge (i_2 = i_3) \wedge (i_1 = i_4) \wedge (\uparrow p_1 = ps_4) \textbf{ then } 4$
$\quad \textbf{else } 0)$

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 :**

$\text{lookup\_udp } socks\ quad\ bound\ arch =$
$\quad \{\mathsf{sid} \mid \mathsf{sid} \in \mathbf{dom}(socks) \wedge$
$\qquad \textbf{let } s = socks[\mathsf{sid}] \textbf{ in}$
$\qquad \textbf{let } sn = \text{match\_score}(s.is_1, s.ps_1, s.is_2, s.ps_2)quad \textbf{ in}$
$\qquad\quad sn > 0 \wedge$
$\qquad\quad \textbf{if } \text{windows\_arch } arch \textbf{ then}$
$\qquad\qquad \textbf{if } sn = 1 \textbf{ then}$
$\qquad\qquad\quad \neg(\exists(sid', s') :: (socks \backslash\backslash \mathsf{sid}).\ \text{match\_score}(s'.is_1, s'.ps_1, s'.is_2, s'.ps_2)quad > sn)$
$\qquad\qquad \textbf{else } \mathbf{T}$
$\qquad\quad \textbf{else}$
$\qquad\qquad \neg(\exists(sid', s') :: (socks \backslash\backslash \mathsf{sid}).$
$\qquad\qquad\qquad (\text{match\_score}(s'.is_1, s'.ps_1, s'.is_2, s'.ps_2)quad > sn \vee$
$\qquad\qquad\qquad (\text{linux\_arch } arch \wedge \text{match\_score}(s'.is_1, s'.ps_1, s'.is_2, s'.ps_2)quad = sn \wedge$
$\qquad\qquad\qquad\quad \text{bound\_after } sid'\ \mathsf{sid}\ bound)))\}$

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

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

Variations

FreeBSD	For FreeBSD, the set contains the sockets for which the score is greater than zero and there is no other socket in *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 :

$\text{tcp\_socket\_best\_match}(socks : \mathsf{sid} \mapsto \mathsf{socket})(\mathsf{sid}, sock)(seg : \mathsf{tcpSegment})\, arch =$
(* is the socket sid the best match for segment seg? *)
$\textbf{let}\ \ s = sock\ \ \textbf{in}$
$\textbf{let}\ \ score = \text{match\_score}(s.is_1, s.ps_1, s.is_2, s.ps_2)$
$\qquad\qquad\qquad\qquad (\textbf{the}\ \ seg.is_1, seg.ps_1, \textbf{the}\ \ seg.is_2, seg.ps_2)\ \textbf{in}$
$\neg(\exists(sid', s') :: socks \backslash\backslash \mathsf{sid}.$
$\qquad \text{match\_score}(s'.is_1, s'.ps_1, s'.is_2, s'.ps_2)$
$\qquad\qquad\qquad\qquad (\textbf{the}\ \ seg.is_1, seg.ps_1, \textbf{the}\ \ seg.is_2, seg.ps_2) > score)$

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

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

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

$\text{lookup\_icmp}\ \ socks\ icmp\ arch\ bound =$
$\quad \{sid0 \mid \exists(\mathsf{sid}, sock) :: socks.$
$\qquad\quad sock.ps_1 = icmp.ps_3 \wedge \text{proto\_of}\ \ sock.pr = icmp.proto \wedge sid0 = \mathsf{sid} \wedge$
$\qquad\quad \textbf{if}\ \ \text{windows\_arch}\ \ arch\ \ \textbf{then}\ \ \mathbf{T}$
$\qquad\quad \textbf{else}$
$\qquad\qquad sock.is_1 = icmp.is_3 \wedge sock.is_2 = icmp.is_4 \wedge$
$\qquad\qquad (sock.ps_2 = icmp.ps_4 \vee$
$\qquad\qquad (\text{linux\_arch}\ \ arch \wedge$
$\qquad\qquad\quad \text{proto\_of}\ \ sock.pr = \text{PROTO\_UDP} \wedge sock.ps_2 = * \wedge$
$\qquad\qquad\quad \neg(\exists(sid', s) :: (socks \backslash\backslash \mathsf{sid}).$
$\qquad\qquad\qquad\quad s.is_1 = icmp.is_3 \wedge s.is_2 = icmp.is_4 \wedge$
$\qquad\qquad\qquad\quad s.ps_1 = icmp.ps_3 \wedge s.ps_2 = icmp.ps_4 \wedge$
$\qquad\qquad\qquad\quad \text{proto\_of}\ \ s.pr = icmp.proto \wedge$
$\qquad\qquad\qquad\quad \text{bound\_after}\ \ sid'\ \mathsf{sid}\ bound)$
$\qquad\qquad))\}$

Description

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

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

Variations

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

12.5 Timers (TCP and UDP)

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

12.5.1 Summary

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

12.5.2 Rules

– **TCP slow timer, typically 500ms resolution (for keepalive, MSL, linger, badrxtwin) :**
slow_timer d = fuzzy_timer d SLOW_TIMER_INTVL SLOW_TIMER_MODEL_INTVL

– **TCP fast timer, typically 200ms resolution (for delack) :**
fast_timer d = fuzzy_timer d FAST_TIMER_INTVL FAST_TIMER_MODEL_INTVL

– **kernel timer, typically 10ms resolution (for timestamp valid, pselect) :**
kern_timer d = fuzzy_timer d KERN_TIMER_INTVL KERN_TIMER_MODEL_INTVL

– **scheduling timer (for OS returns) :**
sched_timer = upper_timer dschedmax

– **in-queue timer (incoming message processing) :**
inqueue_timer = upper_timer diqmax

– **out-queue timer (outgoing message emission) :**
outqueue_timer = upper_timer doqmax

Description

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

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

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

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

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

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

12.6 Time values for socket options (TCP and UDP)

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

12.6.1 Summary

time_of_tltime	convert $(sec, nsec)$ pair to real time value
time_of_tltimeopt	convert optional $(sec, nsec)$ pair to real time value (where $*$ mapped to ∞)
tltimeopt_wf	is an optional $(sec, nsec)$ pair well-formed?
tltimeopt_of_time	convert a time value to an optional $(sec, nsec)$ pair

12.6.2 Rules

– **convert** $(sec, nsec)$ **pair to real time value :**
$(\text{time\_of\_tltime} : \mathsf{int\#int} \rightarrow \mathsf{time})$
$(sec, nsec) = \mathsf{time}(\text{real\_of\_int}\ \ sec + \text{real\_of\_int}\ \ nsec/1000000000)$

– **convert optional** $(sec, nsec)$ **pair to real time value (where** $*$ **mapped to** ∞**) :**
$\text{time\_of\_tltimeopt}\ * = \infty\ \wedge$
$\text{time\_of\_tltimeopt}(\uparrow sn) = \text{time\_of\_tltime}\ \ sn$

– **is an optional** $(sec, nsec)$ **pair well-formed? :**
$(\text{tltimeopt\_wf} : (\mathsf{int\#int})\ \mathsf{option} \rightarrow \mathsf{bool})$
$* = \mathbf{T}\ \wedge$
$\text{tltimeopt\_wf}(\uparrow(sec, nsec)) = (sec \geq 0 \wedge nsec \geq 0 \wedge nsec < 1000000000)$

– **convert a time value to an optional** $(sec, nsec)$ **pair :**
$(\text{tltimeopt\_of\_time} : \mathsf{time} \rightarrow (\mathsf{int\#int})\ \mathsf{option})t$
$= @x.\ \text{tltimeopt\_wf}\ \ x \wedge \text{time\_of\_tltimeopt}\ \ x = t$ (* garbage if t not nonnegative integral number of nsec *)

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

12.7 Queues (TCP and UDP)

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

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

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

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

12.7.1 Summary

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

12.7.2 Rules

– **attempt to enqueue a message :**

enqueue $dq((q)_d, \mathsf{msg}, (q')_{d'}, queued)$
$= ((\text{INFINITE\_RESOURCES} \implies queued) \wedge$
$\quad (q', d') = (\textbf{if } queued \textbf{ then } (q \mathbin{@} [\mathsf{msg}], dq) \textbf{ else } (q, d))$
$)$

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

– **attempt to enqueue onto the in-queue :**

enqueue_iq = enqueue inqueue_timer

– **attempt to enqueue onto the out-queue :**

enqueue_oq = enqueue outqueue_timer

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

– **attempt to dequeue a message :**

dequeue $dq((q)_d, (q')_{d'}, \mathsf{msg})$
$= \textbf{case } q \textbf{ of}$
$\quad (msg0 :: q_0) \rightarrow q' = q_0 \wedge \mathsf{msg} = \uparrow msg0 \wedge d' = (\textbf{if } q_0 = [\,] \textbf{ then } \text{never\_timer } \textbf{else } dq) \parallel$
$\quad [\,] \rightarrow q' = q \wedge \mathsf{msg} = * \wedge d' = d$

– **attempt to dequeue from the in-queue :**

dequeue_iq = dequeue inqueue_timer

– **attempt to dequeue from the out-queue :**

dequeue_oq = dequeue outqueue_timer

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

– **attempt to route and then enqueue an outgoing message :**
route_and_enqueue_oq$(rttab, ifds, oq, \mathsf{msg}, oq', es, arch)$
$= \textbf{case}\ \ \text{test\_outroute}(\mathsf{msg}, rttab, ifds, arch)\ \textbf{of}$
$\quad * \rightarrow \mathbf{F}$
$\| \uparrow(\uparrow e) \rightarrow oq' = oq \wedge es = \uparrow e$
$\| \uparrow * \rightarrow \exists queued.$
$\qquad \text{enqueue\_oq}(oq, \mathsf{msg}, oq', queued) \wedge$
$\qquad es = \textbf{if}\ \ queued\ \ \textbf{then}\ \ *\ \ \textbf{else}\ \uparrow \text{ENOBUFS}$

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

– **attempt to enqueue a list of messages :**
enqueue_list_qinfo $dq(q, (\mathsf{msg}, queued) :: msgqs, q')$
$= (\exists q_0.$
$\quad \text{enqueue}\ \ dq(q, \mathsf{msg}, q_0, queued) \wedge$
$\quad \text{enqueue\_list\_qinfo}\ \ dq(q_0, msgqs, q')) \wedge$
enqueue_list_qinfo $dq(q, [\,], q')$
$= (q' = q)$

– **attempt to enqueue a list of messages, ignoring success flags :**
enqueue_list $dq(q, msgs, q', queued) =$
$(\exists msgqs.$
enqueue_list_qinfo $dq(q, msgqs, q') \wedge$
$msgs = \textbf{map}\ \textbf{fst}\ \ msgqs \wedge$
$queued = \textbf{every}(\lambda x.\, \textbf{snd}\ \ x = \mathbf{T}) msgqs)$

– **attempt to enqueue a list of messages onto the out-queue :**
enqueue_oq_list_qinfo = enqueue_list_qinfo outqueue_timer

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

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

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

– **should an incoming incomplete connection be accepted? :**
accept_incoming_q0$(lis : \mathsf{socket_listen})(b : \mathsf{bool})$
$= (b = \textbf{length}\ \ lis.q < \text{backlog\_fudge}\ \ lis.qlimit)$

– **should an incoming completed connection be accepted? :**
accept_incoming_q$(lis : \mathsf{socket_listen})(b : \mathsf{bool})$
$= (b = \textbf{length}\ \ lis.q < 3 * \text{backlog\_fudge}\ \ lis.qlimit\ \ \textbf{div}\ \ 2)$

– **drop from incomplete-connection queue? :**
drop_from_q0$(lis : \mathsf{socket_listen})(b : \mathsf{bool})$
$= ((\textbf{length}\ \ lis.q_0 \geq \text{TCP\_Q0MINLIMIT} \wedge b = \mathbf{T}) \vee$
$\quad (\textbf{length}\ \ lis.q_0 < \text{TCP\_Q0MAXLIMIT} \wedge b = \mathbf{F}))$

Description A listening socket has two queues, the incomplete connections queue $lis.q_0$ and the completed connections queue $lis.q$. An incoming incomplete (respectively, completed) connection be accepted onto $lis.q_0$ (respectively, $lis.q$) if the relevant queue is not full. Intriguingly, for FreeBSD 4.6-RELEASE, this specification is correct, but if syncaches were to be turned off, the condition in the q_0 case would be **length** $lis.q < 3 * lis.qlimit/2$ instead. Existing incomplete connections may dropped from $lis.q_0$ to make room if its length is between its minimum and maximum limits.

12.8 TCP Options (TCP only)

TCP option handling.

12.8.1 Summary

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

12.8.2 Rules

– **Constrain the TCP timestamp option values that appear in an outgoing segment :**
do_tcp_options *cb_tf_doing_tstmp cb_ts_recent cb_ts_val* =
if *cb_tf_doing_tstmp* **then**
  let ts_ecr' = **option_case** (ts_seq $0w$) **I** (timewindow_val_of *cb_ts_recent*) **in**
    $\uparrow(cb\_ts\_val, ts\_ecr')$
else
  $*$

– **Calculate the length consumed by the TCP options in a real TCP segment :**
calculate_tcp_options_len *cb_tf_doing_tstmp* =
if *cb_tf_doing_tstmp* **then** 12 **else** 0 : num

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

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

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

12.9.1 Summary

calculate_buf_sizes	Calculate buffer sizes for *rcvbufsize*, *sndbufsize*, *t_maxseg*, and *snd_cwnd*
calculate_bsd_rcv_wnd	Calculation of *rcv_wnd*
send_queue_space	

Rule version: $Id: TCP1_auxFnsScript.sml,v 1.219 2005/03/17 11:35:34 kw217 Exp $

12.9.2 Rules

– **Calculate buffer sizes for** *rcvbufsize*, *sndbufsize*, *t_maxseg*, **and** *snd_cwnd* **:**
calculate_buf_sizes *cb_t_maxseg seg_mss bw_delay_product_for_rt is_local_conn*
rcvbufsize sndbufsize cb_tf_doing_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**

(* compute initial cwnd *)
let $snd\_cwnd = t\_maxseg''' *$ (**if** *is_local_conn* **then** SS_FLTSZ_LOCAL **else** SS_FLTSZ) **in**
$(rcvbufsize'', sndbufsize'', t\_maxseg''', snd\_cwnd)$

Description Used in *deliver_in_1* and *deliver_in_2*.

– **Calculation of** *rcv_wnd* **:**
calculate_bsd_rcv_wnd *sf tcp_sock* =
max(**num**(*tcp_sock.cb.rcv_adv* − *tcp_sock.cb.rcv_nxt*))
(*sf.n*(SO_RCVBUF) − **length** *tcp_sock.rcvq*)

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

– :
send_queue_space$(sndq_max : \mathsf{num}) sndq_size\ oob\ arch\ maxseg\ i_2 =$
$\{n \mid$ **if** bsd_arch *arch* **then**
$n \le (sndq\_max - sndq\_size) + ($**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 $n \ge 0\}$

Description Calculation of the usable send queue space.

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

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

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

12.10 Band limiting (TCP and UDP)

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

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

12.10.1 Summary

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

12.10.2 Rules

– initial state of bandlimiter :
bandlim_state_init = [ ] : *bandlim_state*

– the trivial 'always OK' bandlimiter :
(bandlim_rst_ok_always : tcpSegment # ts_seq # bandlim_reason # *bandlim_state* → bool # *bandlim_state*)
$(seg, ticks, reason, bndlm)$
$=$ **let** $bndlm' = (seg, ticks, reason) :: bndlm$
in
$(\mathbf{T}, bndlm')$

– simple-bandlimiter rate settings :

$$
\begin{array}{l}
(\text{simple\_limit} : \mathsf{bandlim\_reason} \to \mathsf{num\ \ option}) \\
\qquad\qquad \text{BANDLIM\_UNLIMITED} = * \wedge \\
\text{simple\_limit}\ \ \text{BANDLIM\_RST\_CLOSEDPORT} = \uparrow 200 \wedge \\
\text{simple\_limit}\ \ \text{BANDLIM\_RST\_OPENPORT} = \uparrow 200
\end{array}
$$

– **a simple rate-limiting bandlimiter :**

$$
\begin{array}{l}
(\text{bandlim\_rst\_ok\_simple} : \mathsf{tcpSegment}\#\,\text{ts\_seq}\,\#\mathsf{bandlim\_reason}\#\mathit{bandlim\_state} \to \mathsf{bool}\#\mathit{bandlim\_state}) \\
\qquad\qquad (\mathit{seg}, \mathit{ticks}, \mathit{reason}, \mathit{bndlm}) \\
= \textbf{let}\ \ \mathit{reasoneq} = (\lambda r_0.\lambda(s,t,r).r = r_0) \\
\quad \textbf{and}\ \ \mathit{ticksgt} = (\lambda t_0.\lambda(s,t,r).t > t_0) \\
\quad \textbf{in} \\
\quad \textbf{let}\ \ \mathit{count} = \textbf{length}(\textbf{filter}(\mathit{reasoneq}\ \mathit{reason})(\text{TAKEWHILE}(\mathit{ticksgt}(\mathit{ticks} - \text{num\_floor}(1 * \text{HZ})))\mathit{bndlm})) \\
\quad \textbf{in} \\
\quad ((\textbf{case}\ \ \text{simple\_limit}\ \ \mathit{reason}\ \ \textbf{of} \\
\qquad\quad * \to \mathbf{T} \\
\qquad \| \uparrow n \to \mathit{count} < n), \\
\quad (\mathit{seg}, \mathit{ticks}, \mathit{reason}) :: \mathit{bndlm})
\end{array}
$$

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

– **the bandlimiter actually used :**

bandlim_rst_ok = bandlim_rst_ok_simple

Description Which band limiter to use?

– **enqueue onto out-queue if allowed by bandlimiter :**

$$
\begin{array}{l}
\text{enqueue\_oq\_bndlim\_rst}(\mathit{oq}, \mathit{seg}, \mathit{ticks}, \mathit{reason}, \mathit{bndlm}, \mathit{oq}', \mathit{bndlm}', \mathit{queued\_or\_dropped}) \\
= \textbf{let}\ (\mathit{emit}, \mathit{bndlm}_0) = \text{bandlim\_rst\_ok}(\mathit{seg}, \mathit{ticks}, \mathit{reason}, \mathit{bndlm}) \\
\textbf{in} \\
\mathit{bndlm}' = \mathit{bndlm}_0 \wedge \\
\textbf{if}\ \ \mathit{emit}\ \ \textbf{then} \\
\quad \text{enqueue\_oq}(\mathit{oq}, \text{TCP}\ \mathit{seg}, \mathit{oq}', \mathit{queued\_or\_dropped}) \\
\textbf{else} \\
\quad (\mathit{oq}' = \mathit{oq} \wedge \mathit{queued\_or\_dropped} = \mathbf{T})
\end{array}
$$

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

12.11 UDP support (UDP only)

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

12.11.1 Summary

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

12.11.2 Rules

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

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

Description For use in UDP *sendto*().

12.12 TCP timing and RTT (TCP only)

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

12.12.1 Summary

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

12.12.2 Rules

– select this architecture's retransmit backoff list :

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

Rule version: $Id: TCP1_auxFnsScript.sml,v 1.219 2005/03/17 11:35:34 kw217 Exp $

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

– obtain the mode of a backoff timer :
(mode_of : (rexmtmode#num)timed option → rexmtmode option)
$(\uparrow(((mode, \_))\_)) = \uparrow mode \wedge$
mode_of $* = *$

– obtain the shift of a backoff timer :
shift_of$(\uparrow(((_, shift))_)) = shift$

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

– compute retransmit timeout to use :
computed_rto(*backoffs* : num list)(*shift* : num)(*ri* : rttinf)
= **real_of_num**(*EL shift backoffs*) $*$
max $ri.t\_rttmin(ri.t\_srtt + 4 * ri.t\_rttvar)$

– compute the last-used *rxtcur* :
computed_rxtcur(*ri* : rttinf)(*arch* : *arch*)
= **max** $ri.t\_rttmin$
(**min**(**the** TCPTV_REXMTMAX)
(computed_rto(**if** $ri.t\_wassyn$ **then** tcp_syn_backoffs *arch*
else tcp_backoffs *arch*)
$ri.t\_lastshift$ *ri*))

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

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

Note: TCPTV_REXMTMAX had better not be infinite!

– construct retransmit timer (generic) :
start_tt_rexmt_gen(*mode* : rexmtmode)(*backoffs* : num list)(*shift* : num)(*wantmin* : bool)(*ri* : rttinf)
= **let** *rxtcur* = **max**(**if** *wantmin*
then max $ri.t\_rttmin(ri.t\_lastrtt + 2/\,\mathrm{HZ})$
else $ri.t\_rttmin$)
(**min**(**the** TCPTV_REXMTMAX (* better not be infinite! *))
(computed_rto *backoffs shift ri*)
)

in
$\uparrow(((mode, shift))_{\text{slow\_timer}(\mathsf{time}\ rxtcur)})$

– **construct normal retransmit timer :**
start_tt_rexmt($arch$: $arch$) = start_tt_rexmt_gen Rexmt(tcp_backoffs $arch$)

– **construct** SYN**-retransmit timer :**
start_tt_rexmtsyn($arch$: $arch$) = start_tt_rexmt_gen RexmtSyn(tcp_syn_backoffs $arch$)

– **construct persist timer :**
start_tt_persist($shift$: num)(ri : rttinf)($arch$: $arch$)
= **let** cur = **max**(**the** TCPTV_PERSMIN (* better not be infinite! *))
(**min**(**the** TCPTV_PERSMAX (* better not be infinite! *))
(computed_rto(tcp_backoffs $arch$) $shift$ ri)
)
in
$\uparrow(((\text{Persist}, shift))_{\text{slow\_timer}(\mathsf{time}\ cur)})$

Description

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

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

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

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

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

(* t_softerror, t_rttseg, and t_rxtcur must be handled by the caller *)
$\rrbracket$

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

– **expand congestion window :**
expand_cwnd *ssthresh maxseg maxwin cwnd*
$= \textbf{min}\ \mathit{maxwin}(\mathit{cwnd} + (\textbf{if}\ \ \mathit{cwnd} > \mathit{ssthresh}\ \ \textbf{then}\ (\mathit{maxseg} * \mathit{maxseg})\,\textbf{div}\ \ \mathit{cwnd}\ \ \textbf{else}\ \ \mathit{maxseg}))$

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

12.13 Path MTU Discovery (TCP only)

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

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

12.13.1 Summary

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

12.13.2 Rules

– **find next-smaller element of a set :**
$(\mathrm{next\_smaller} : (\mathsf{num} \rightarrow \mathsf{bool}) \rightarrow \mathsf{num} \rightarrow \mathsf{num}) xs\ y = @x :: xs.x < y \wedge \forall x' :: xs.x' > x \implies x' \geq y$

– **path MTU plateaus to try :**
mtu_tab *arch* = **if** linux_arch *arch* **then**
$\{32000; 17914; 8166; 4352; 2002; 1492; 576; 296; 216; 128; 68\} : \mathsf{num\ set}$
else
$\{65535; 32000; 17914; 8166; 4352; 2002; 1492; 1006; 508; 296; 68\}$

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

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

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

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

12.14 Reassembly (TCP only)

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

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

12.14.1 Summary

tcp_reass	perform TCP segment reassembly
tcp_reass_prune	drop prefix of reassembly queue

12.14.2 Rules

– **perform TCP segment reassembly :**

$$
\begin{array}{l}
\text{tcp\_reass}\ \mathit{seq}(\mathit{rsegq} : \mathsf{tcpReassSegment\ list}) = \\
\textbf{let}\ \mathit{myrel} = \{(i, c) \mid \exists \mathit{rseg}. \\
\qquad\qquad \mathit{rseg} \in \mathit{rsegq} \land \\
\qquad\qquad i \geq \mathit{rseg.seq} \land \\
\qquad\qquad i < \mathit{rseg.seq} + \textbf{length}\ \mathit{rseg.data} + \\
\qquad\qquad\quad (\textbf{if}\ \mathit{rseg.spliced\_urp} \neq * \ \textbf{then}\ 1\ \textbf{else}\ 0) \land \\
\qquad\qquad (\textbf{case}\ \mathit{rseg.spliced\_urp}\ \textbf{of} \\
\qquad\qquad\quad \uparrow(n) \rightarrow \\
\qquad\qquad\qquad (\textbf{if}\ i > n\ \textbf{then} \\
\qquad\qquad\qquad\quad c = \uparrow(\mathit{EL}(\textbf{num}(i - \mathit{rseg.seq} - 1))(\mathit{rseg.data})) \\
\qquad\qquad\qquad\ \textbf{else if}\ i = n\ \textbf{then} \\
\qquad\qquad\qquad\quad c = * \\
\qquad\qquad\qquad\ \textbf{else} \\
\qquad\qquad\qquad\quad c = \uparrow(\mathit{EL}(\textbf{num}(i - \mathit{rseg.seq}))(\mathit{rseg.data}))) \parallel \\
\qquad\qquad\quad * \rightarrow \\
\qquad\qquad\qquad\quad c = \uparrow(\mathit{EL}(\textbf{num}(i - \mathit{rseg.seq}))(\mathit{rseg.data})))\}\ \textbf{in} \\
\{(\mathit{cs}', \mathit{len}, \mathit{FIN}) \mid \exists \mathit{cs}.\mathit{cs}' = \text{CONCAT\_OPTIONAL}\ \mathit{cs} \land \\
\qquad (\forall n : \mathit{num}.n < \textbf{length}\ \mathit{cs} \implies (\mathit{seq} + n, \mathit{EL}\ n\ \mathit{cs}) \in \mathit{myrel}) \land \\
\qquad (\neg\exists c.(\mathit{seq} + \textbf{length}\ \mathit{cs}, c) \in \mathit{myrel}) \land \\
\qquad (\mathit{len} = \textbf{length}\ \mathit{cs}) \land \\
\qquad (\mathit{FIN} = \exists \mathit{rseg}.\mathit{rseg} \in \mathit{rsegq} \land \\
\qquad\qquad \mathit{rseg.seq} + \textbf{length}\ \mathit{rseg.data} + \\
\qquad\qquad\quad (\textbf{if}\ \mathit{rseg.spliced\_urp} \neq * \ \textbf{then}\ 1\ \textbf{else}\ 0) = \\
\qquad\qquad\qquad \mathit{seq} + \textbf{length}\ \mathit{cs} \land \\
\qquad\qquad \mathit{rseg.FIN})\}
\end{array}
$$

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

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

– **drop prefix of reassembly queue :**
tcp_reass_prune $seq(rsegq : \mathsf{tcpReassSegment\ list})$ =
$\mathbf{filter}(\lambda rseg.rseg.seq + \mathbf{length}\ \ rseg.data + (\mathbf{if}\ \ rseg.spliced\_urp \neq * \ \ \mathbf{then}\ \ 1\ \ \mathbf{else}\ \ 0) +$
$(\mathbf{if}\ \ rseg.FIN\ \ \mathbf{then}\ \ 1\ \ \mathbf{else}\ \ 0) > seq)rsegq$

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

12.15 The initial TCP control block (TCP only)

The initial state of the TCP control block.

12.15.1 Summary

initial_cb

12.15.2 Rules

– **:**
initial_cb =
$\langle\!|\ t\_segq := [\,];$
$tt\_rexmt := *;$
$tt\_keep := *;$
$tt\_2msl := *;$
$tt\_delack := *;$
$tt\_conn\_est := *;$
$tt\_fin\_wait\_2 := *;$
$tf\_needfin := \mathbf{F};$
$tf\_shouldacknow := \mathbf{F};$
$snd\_una := \mathrm{tcp\_seq\_local}\ \ 0w;$
$snd\_max := \mathrm{tcp\_seq\_local}\ \ 0w;$
$snd\_nxt := \mathrm{tcp\_seq\_local}\ \ 0w;$
$snd\_wl1 := \mathrm{tcp\_seq\_foreign}\ \ 0w;$
$snd\_wl2 := \mathrm{tcp\_seq\_local}\ \ 0w;$
$iss := \mathrm{tcp\_seq\_local}\ \ 0w;$
$snd\_wnd := 0;$
$snd\_cwnd := \mathrm{TCP\_MAXWIN} \ll \mathrm{TCP\_MAXWINSCALE};$
$snd\_ssthresh := \mathrm{TCP\_MAXWIN} \ll \mathrm{TCP\_MAXWINSCALE};$
$rcv\_wnd := 0;$
$tf\_rxwin0sent := \mathbf{F};$
$rcv\_nxt := \mathrm{tcp\_seq\_foreign}\ \ 0w;$
$rcv\_up := \mathrm{tcp\_seq\_foreign}\ \ 0w;$
$irs := \mathrm{tcp\_seq\_foreign}\ \ 0w;$
$rcv\_adv := \mathrm{tcp\_seq\_foreign}\ \ 0w;$
$snd\_recover := \mathrm{tcp\_seq\_local}\ \ 0w;$
$t\_maxseg := \mathrm{MSSDFLT};$
$t\_advmss := *;$
$t\_rttseg := *;$
$t\_rttinf :=$
$\langle\!|$

$t\_rttupdated := 0;$
$tf\_srtt\_valid := \mathbf{F};$
$t\_srtt := \mathrm{TCPTV\_RTOBASE};$
$t\_rttvar := \mathrm{TCPTV\_RTTVARBASE};$
$t\_rttmin := \mathrm{TCPTV\_MIN};$
$t\_lastrtt := 0;$
$t\_lastshift := 0;$
$t\_wassyn := \mathbf{F}$(* if t_lastshift=0, this doesn't make a difference *)
$\rrbracket;$
$t\_dupacks := 0;$
$t\_idletime := \mathrm{stopwatch\_zero};$
$t\_softerror := *;$
$snd\_scale := 0;$
$rcv\_scale := 0;$
$request\_r\_scale := *;$(* this like many other things is overwritten with the chosen value later - cf tcp_newtcpcb() *)
$tf\_doing\_ws := \mathbf{F};$
$ts\_recent := \textsc{TimeWindowClosed};$
$tf\_req\_tstmp := \mathbf{F};$ (* cf tcp_newtcpcb() *)
$tf\_doing\_tstmp := \mathbf{F};$
$last\_ack\_sent := \mathrm{tcp\_seq\_foreign}\ 0w;$
$bsd\_cantconnect := \mathbf{F};$
$snd\_cwnd\_prev := 0;$
$snd\_ssthresh\_prev := 0;$
$t\_badrxtwin := \textsc{TimeWindowClosed}$
(* Note: everything should be listed here, leaving nothing as ARB. *)
(* Many are always overwritten, however. *)
$\rrbracket$

Chapter 13

Relational monad

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

13.1 Relational monad (TCP only)

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

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

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

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

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

Note that *outsegs*, the list of messages, is actually a list of arbitrary type; this enables us to lift the glue to the type $\mathsf{msg\#bool}$ in *deliver_in_3*, where we need the flag to deal with queueing failure.

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

13.1.1 Summary

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

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

13.1.2 Rules

– **normal sequencing :**
$(op1 \ \text{andThen} \ op2) =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad \exists sock_1 \ bndlm_1 \ outsegs_1 \ continue_1 \ sock_2 \ bndlm_2 \ outsegs_2 \ continue_2.$
$\quad op1(sock, bndlm)((sock_1, bndlm_1, outsegs_1), continue_1) \wedge$
$\quad \textbf{if} \ continue_1 \ \textbf{then}$
$\qquad op2(sock_1, bndlm_1)((sock_2, bndlm_2, outsegs_2), continue_2) \wedge$
$\qquad (sock' = sock_2 \wedge bndlm' = bndlm_2 \wedge outsegs' = outsegs_1 \ @ \ outsegs_2 \wedge continue' = continue_2)$
$\quad \textbf{else}$
$\qquad (sock' = sock_1 \wedge bndlm' = bndlm_1 \wedge outsegs' = outsegs_1 \wedge continue' = \mathbf{F})$

– **do nothing, and continue (unit for** andThen**) :**
$\text{cont} =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad (sock' = sock \wedge bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{T})$

– **do nothing, and stop (zero for** andThen**) :**
$\text{stop} =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad (sock' = sock \wedge bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{F})$

– **assert truth of condition, and continue :**
$\text{assert} \ p =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad (p \wedge sock' = sock \wedge bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{T})$

– **assertion violated; fail noisily :**
$assert\_failure \ s =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad ASSERTION\_FAILURE \ s$

– **choose a value from a set, nondeterministically :**
$\text{chooseM} \ s \ f =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad \textbf{choose} \ x :: s.f \ x(sock, bndlm)((sock', bndlm', outsegs'), continue')$

– **get current socket :**
$\text{get\_sock} \ f =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad f \ sock(sock, bndlm)((sock', bndlm', outsegs'), continue')$

– **assert current socket is TCP, and get its protocol data :**
$\text{get\_tcp\_sock} \ f =$
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : {'msg} \ \mathsf{list}), continue' : \mathsf{bool}).$
$\quad \exists tcp\_sock.$
$\quad sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\quad f \ tcp\_sock(sock, bndlm)((sock', bndlm', outsegs'), continue')$

– **assert current socket is TCP, and get its control block :**
get_cb f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$\exists tcp\_sock.$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$f\ tcp\_sock.cb(sock, bndlm)((sock', bndlm', outsegs'), continue')$

– **apply function to current socket :**
modify_sock f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(sock' = f\ sock \wedge bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{T})$

– **apply function to current socket :**
modify_tcp_sock f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(\exists tcp\_sock.$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock' = sock\ \langle\!\!\langle\ pr := \mathrm{TCP\_PROTO}(f\ tcp\_sock)\rangle\!\!\rangle \wedge bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{T})$

– **assert current socket is TCP, and apply function to its control block :**
modify_cb f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$\exists tcp\_sock.$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$(sock' = sock\ \langle\!\!\langle\ pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \langle\!\!\langle\ cb := (f\ tcp\_sock.cb)\rangle\!\!\rangle)\rangle\!\!\rangle \wedge$
$bndlm' = bndlm \wedge outsegs' = [\,] \wedge continue' = \mathbf{T})$

– **append segments to current output list :**
emit_segs $segs$ =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(sock' = sock \wedge bndlm' = bndlm \wedge outsegs' = segs \wedge continue' = \mathbf{T})$

– **append segments specified by a predicate (nondeterministic) :**
emit_segs_pred f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(sock' = sock \wedge f\ bndlm\ bndlm'\ outsegs' \wedge continue' = \mathbf{T})$

– **lift a monadic operation not involving** *continue* **or** *bndlm* **:**
mliftc f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(f\ sock(sock', outsegs') \wedge bndlm' = bndlm \wedge continue' = \mathbf{T})$

– **lift a monadic operation not involving** *continue* **:**
mliftc_bndlm f =
$\lambda(sock : \mathsf{socket}, bndlm : bandlim\_state)((sock' : \mathsf{socket}, bndlm' : bandlim\_state, outsegs' : 'msg\ \mathsf{list}), continue' : \mathsf{bool}).$
$(f(sock, bndlm)(sock', bndlm', outsegs') \wedge continue' = \mathbf{T})$

Chapter 14

Auxiliary functions for TCP segment creation and drop

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

14.1 SYN and RST Segment Creation (TCP only)

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

14.1.1 Summary

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

14.1.2 Rules

– **Make a SYN segment for emission by** *connect_1* **etc :**
make_syn_segment $cb(i_1, i_2, p_1, p_2) ts\_val\ seg' =$
(**choose** $urp\_any :: UNIV$.
choose $ack\_any :: UNIV$.

(* Determine window size; fail if out of range *)
let $win =$ **n2w** $cb.rcv\_wnd$ **in**
w2n $win = cb.rcv\_wnd \wedge$

(* Choose a window scaling; fail if out of range *)
(* Note there may be a better place for this assertion. *)
let $ws =$ **option_map** $CHR\ cb.request\_r\_scale$ **in**
(**is_some** $cb.request\_r\_scale \implies$ **ord**(**the** ws) $=$ **the** $cb.request\_r\_scale) \wedge$
(**case** ws **of** $* \to$ **T** $\parallel \uparrow n \to$ **ord** $n \leq$ TCP_MAXWINSCALE) $\wedge$

(* Determine maximum segment size; fail if out of range *)
(* Put the MSS we initially advertise into t_advmss *)
$\textbf{let}\ \ mss = (\textbf{case}\ \ cb.t\_advmss\ \ \textbf{of}$
$\quad * \rightarrow *$
$\quad \| \uparrow v \rightarrow \uparrow(\textbf{n2w}\ \ v))\ \textbf{in}$
$(\textbf{case}\ \ cb.t\_advmss\ \ \textbf{of}$
$\quad * \rightarrow \mathbf{T}$
$\ \| \uparrow v \rightarrow v = \textbf{w2n}(\textbf{the}\ \ mss)) \wedge$

(* Do timestamping? *)
$\textbf{let}\ \ ts = \text{do\_tcp\_options}\ \ cb.tf\_req\_tstmp\ cb.ts\_recent\ ts\_val\ \ \textbf{in}$

$seg' = \langle\!|\ is_1 := \uparrow i_1;$
$\quad is_2 := \uparrow i_2;$
$\quad ps_1 := \uparrow p_1;$
$\quad ps_2 := \uparrow p_2;$
$\quad seq := cb.iss;$
$\quad ack := ack\_any;$
$\quad URG := \mathbf{F};$
$\quad ACK := \mathbf{F};$
$\quad PSH := \mathbf{F};$
$\quad RST := \mathbf{F};$
$\quad SYN := \mathbf{T};$
$\quad FIN := \mathbf{F};$
$\quad win := win;$
$\quad ws := ws;$
$\quad urp := urp\_any;$
$\quad mss := mss;$
$\quad ts := ts;$
$\quad data := [\,]$
$\quad |\!\rangle$
)

– **Make a SYN,ACK segment for emission by** *deliver_in_1*, *deliver_in_2*, **etc.:**
$\text{make\_syn\_ack\_segment}\ \ cb(i_1, i_2, p_1, p_2) ts\_val'\ seg' =$
$\textbf{choose}\ \ urp\_any :: UNIV.$

(* Determine window size; fail if out of range *)
(* We don't scale yet ($\gg$ *rcv_scale'*). RFC1323 says: segments with SYN are not scaled, and BSD agrees. Even though we know what scaling the other end wants to use, and we know whether we are doing scaling, we can't use it until we reach the ESTABLISHED state. *)
$\textbf{let}\ \ win = \textbf{n2w}\ \ cb.rcv_wnd\ \ \textbf{in}\ \ $ (* $rcv\_window - \textbf{length}\ \ data'$ *)
$\textbf{w2n}\ \ win = cb.rcv\_wnd \wedge$

(* If doing window scaling, set it; fail if out of range *)
$\textbf{let}\ \ ws = \textbf{if}\ \ cb.tf\_doing\_ws\ \ \textbf{then}\ \ \uparrow(CHR\ cb.rcv\_scale)\ \textbf{else}\ \ *\ \ \textbf{in}$
$(cb.tf\_doing\_ws \implies \textbf{ord}(\textbf{the}\ \ ws) = cb.rcv\_scale) \wedge$

(* Determine maximum segment size; fail if out of range *)
(* Put the MSS we initially advertise into t_advmss *)
$\textbf{let}\ \ mss = (\textbf{case}\ \ cb.t\_advmss\ \ \textbf{of}$
$\quad * \rightarrow *$
$\quad \| \uparrow v \rightarrow \uparrow(\textbf{n2w}\ \ v))\ \textbf{in}$
$(\textbf{case}\ \ cb.t\_advmss\ \ \textbf{of}$
$\quad * \rightarrow \mathbf{T}$
$\ \| \uparrow v \rightarrow v = \textbf{w2n}(\textbf{the}\ \ mss)) \wedge$

(* Set timestamping option? *)
let ts = do_tcp_options $cb.tf\_doing\_tstmp$ $cb.ts\_recent$ ts_val' **in**

$seg' = \llparenthesis\ is_1 := \uparrow i_1;$
$is_2 := \uparrow i_2;$
$ps_1 := \uparrow p_1;$
$ps_2 := \uparrow p_2;$
$seq := cb.iss;$
$ack := cb.rcv\_nxt;$
$URG := \mathbf{F};$
$ACK := \mathbf{T};$
$PSH := \mathbf{F};$ (* see below *)
$RST := \mathbf{F};$
$SYN := \mathbf{T};$
$FIN := \mathbf{F};$ (* Note: we are not modelling T/TCP *)
$win := win;$
$ws := ws;$
$urp := urp\_any;$
$mss := mss;$
$ts := ts;$
$data := [\,]$ (* see below *)
$\rrparenthesis$

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

– **Make a plain boring ACK segment in response to a SYN,ACK segment :**
make_ack_segment cb $FIN(i_1, i_2, p_1, p_2) ts\_val'$ seg' =
((* SB thinks these should be unconstrained. *)
choose $urp\_garbage :: UNIV$.

(* Determine window size; fail if out of range *)
(* Connection is now established so any scaling should be taken into account *)
(* Note it might be appropriate to clip the value to be in range rather than failing if out of range. *)
let win = **n2w**$(cb.rcv\_wnd \gg cb.rcv\_scale)$ **in**
w2n $win = cb.rcv\_wnd \gg cb.rcv\_scale \wedge$

(* Set timestamping option? *)
let ts = do_tcp_options $cb.tf\_doing\_tstmp$ $cb.ts\_recent$ ts_val' **in**

$seg' = \llparenthesis\ is_1 := \uparrow i_1;$
$is_2 := \uparrow i_2;$
$ps_1 := \uparrow p_1;$
$ps_2 := \uparrow p_2;$
$seq :=$ **if** FIN **then** $cb.snd\_una$ **else** $cb.snd\_nxt;$
$ack := cb.rcv\_nxt;$
$URG := \mathbf{F};$
$ACK := \mathbf{T};$
$PSH := \mathbf{F};$ (* see comment for make_syn_ack_segment *)
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win;$
$ws := *;$
$urp := urp\_garbage;$

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

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

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

bsd_make_phantom_segment $cb(i_1, i_2, p_1, p_2) ts\_val'$ $cantsndmore$ seg' $=$
(**choose** $urp\_garbage :: UNIV.$

(* Determine window size; fail if out of range *)
(* Connection is now established so any scaling should be taken into account *)
(* Note it might be appropriate to clip the value to be in range rather than failing if out of range. *)
let $win = \mathbf{n2w}(cb.rcv\_wnd \gg cb.rcv\_scale)$ **in**
w2n $win = cb.rcv\_wnd \gg cb.rcv\_scale \wedge$

let $FIN = (cantsndmore \wedge cb.snd\_una < (cb.snd\_max - 1))$ **in**

(* Set timestamping option? *)
let $ts =$ do_tcp_options $cb.tf\_doing\_tstmp$ $cb.ts\_recent$ ts_val' **in**

$seg' = \{\!|\ is_1 := \uparrow i_1;$
$is_2 := \uparrow i_2;$
$ps_1 := \uparrow p_1;$
$ps_2 := \uparrow p_2;$
$seq :=$ **if** FIN **then** $cb.snd\_una$ **else** $cb.snd\_max$; (* no flags, no data, and no persist timer so use snd_max *)
$ack := cb.rcv\_nxt$; (* yes, really, even though $\neg ACK$ *)
$URG := \mathbf{F};$
$ACK := \mathbf{F};$
$PSH := \mathbf{F};$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win;$
$ws := *;$
$urp := urp\_garbage;$
$mss := *;$
$ts := ts;$
$data := [\,]$ (* sndq always empty in this situation *)
$|\!\})$

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

make_rst_segment_from_cb $cb(i_1, i_2, p_1, p_2) seg'$ $=$
(* Deliberately unconstrained *)
choose $urp\_garbage :: UNIV.$
choose $URG\_garbage :: UNIV.$
choose $PSH\_garbage :: UNIV.$
choose $win\_garbage :: UNIV.$
choose $data\_garbage :: UNIV.$
choose $FIN\_garbage :: UNIV.$

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

$seg' = \llparenthesis\ is_1 := \uparrow i_1;$
$ps_1 := \uparrow p_1;$
$is_2 := \uparrow i_2;$
$ps_2 := \uparrow p_2;$
$seq := cb.snd\_nxt;$ (* from RFC793p62 *)
$ack := cb.rcv\_nxt;$ (* seems the right thing to do *)
$URG := URG\_garbage;$ (* expect: **F** *)
$ACK := \mathbf{T};$ (* from TCPv1p248 *)
$PSH := PSH\_garbage;$ (* expect: **F** *)
$RST := \mathbf{T};$
$SYN := \mathbf{F};$
$FIN := FIN\_garbage;$ (* expect: **F** *)
$win := win\_garbage;$ (* expect: $0w$ *)
$ws := *;$
$urp := urp\_garbage;$ (* expect: $0w$ *)
$mss := *;$
$ts := *;$ (* RFC1323 S4.2 recommends no TS on RST, and BSD follows this *)
$data := data\_garbage$ (* expect: [ ] *)
$\rrparenthesis$

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

make_rst_segment_from_seg $seg\ seg' =$
$(seg.RST = \mathbf{F} \wedge$ (* Sanity check: never RST a RST *)

$(\exists ack'.$
(* Deliberately unconstrained *)
choose $urp\_garbage :: UNIV.$
choose $URG\_garbage :: UNIV.$
choose $PSH\_garbage :: UNIV.$
choose $win\_garbage :: UNIV.$
choose $data\_garbage :: UNIV.$
choose $FIN\_garbage :: UNIV.$

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

(* Sequence number is zero for RST+ACK segments, otherwise it is the next sequence number expected *)
let $seq' =$ **if** $seg.ACK$ **then** $tcp\_seq\_flip\_sense\ seg.ack$
else tcp_seq_local $0w$ **in**

(**if** ACK' **then**
(* RFC794 S3.4: for RST+ACK segments the ack value must be valid *)
$ack' = tcp\_seq\_flip\_sense\ seg.seq + \mathbf{length}\ seg.data + (\mathbf{if}\ seg.SYN\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$
else
(* otherwise it can be arbitrary, although it possibly should be zero *)
$ack' \in \{n \mid \mathbf{T}\}$
$) \wedge$
$seg' = \llparenthesis\ is_1 := seg.is_2;$
$ps_1 := seg.ps_2;$
$is_2 := seg.is_1;$
$ps_2 := seg.ps_1;$
$seq := seq';$

$ack := ack'$;
$URG := URG\_garbage$; (* expect: **F** *)
$ACK := ACK'$;
$PSH := PSH\_garbage$; (* expect: **F** *)
$RST :=$ **T**;
$SYN :=$ **F**;
$FIN := FIN\_garbage$; (* expect: **F** *)
$win := win\_garbage$; (* expect: $0w$ *)
$ws := *$;
$urp := urp\_garbage$; (* expect: $0w$ *)
$mss := *$;
$ts := *$; (* RFC1323 S4.2 recommends no TS on RST, and BSD follows this *)
$data := data\_garbage$ (* expect: [ ] *)
⟧
))

14.2 General Segment Creation (TCP only)

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

14.2.1 Summary

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

14.2.2 Rules

– **determine whether TCP output is required :**
tcp_output_required $arch$ $ifds_0$ $sock$ =
let tcp_sock = tcp_sock_of $sock$ **in**
let $cb = tcp\_sock.cb$ **in**

(* Note this does not deal with `TF_LASTIDLE` and `PRU_MORETOCOME` *)
let snd_cwnd' =
if $\neg(cb.snd\_max = cb.snd\_una \wedge$
stopwatch_val_of $cb.t\_idletime \geq$ computed_rxtcur $cb.t\_rttinf$ $arch)$
then (* inverted so this clause is tried first *)
$cb.snd\_cwnd$
else
(* The connection is idle and has been for >= 1 RTO *)
(* Reduce snd_cwnd to commence slow start *)
$cb.t\_maxseg *$ (**if** is_localnet $ifds_0$(**the** $sock.is_2$) **then** SS_FLTSZ_LOCAL **else** SS_FLTSZ) **in**

(* Calculate the amount of unused send window *)
let win = **min** $cb.snd\_wnd$ snd_cwnd' **in**
let snd_wnd_unused = **int_of_num** $win - (cb.snd\_nxt - cb.snd\_una)$ **in**

(* Is it possible that a FIN may need to be sent? *)
let $fin_required = (sock.cantsndmore \wedge tcp_sock.st \notin \{$FIN_WAIT_2; TIME_WAIT$\})$ **in**

(* Under BSD, we may need to send a *FIN* in state SYN_SENT or SYN_RECEIVED, so we may effectively still have a *SYN* on the send queue. *)

let $syn\_not\_acked = (\mathrm{bsd\_arch}\ arch \wedge tcp\_sock.st\ \in \{\mathrm{SYN\_SENT}; \mathrm{SYN\_RECEIVED}\})$ **in**

(* Is there data or a FIN to transmit? *)
let $last\_sndq\_data\_seq = cb.snd\_una + \mathbf{length}\ tcp\_sock.sndq$ **in**
let $last\_sndq\_data\_and\_fin\_seq = last\_sndq\_data\_seq + (\mathbf{if}\ fin\_required\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$
$+ (\mathbf{if}\ syn\_not\_acked\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$ **in**
let $have\_data\_to\_send = cb.snd\_nxt < last\_sndq\_data\_seq$ **in**
let $have\_data\_or\_fin\_to\_send = cb.snd\_nxt < last\_sndq\_data\_and\_fin\_seq$ **in**

(* The amount by which the right edge of the advertised window could be moved *)
let $window\_update\_delta = (int\_min(\mathbf{int\_of\_num}(\mathrm{TCP\_MAXWIN} \ll cb.rcv\_scale))$
$(\mathbf{int\_of\_num}(sock.sf.n(\mathrm{SO\_RCVBUF})) - \mathbf{int\_of\_num}(\mathbf{length}$
$tcp\_sock.rcvq))) -$
$(cb.rcv\_adv - cb.rcv\_nxt)$ **in**

(* Send a window update? This occurs when (a) the advertised window can be increased by at least two maximum segment sizes, or (b) the advertised window can be increased by at least half the receive buffer size. See `tcp_output.c:322ff`. *)
let $need\_to\_send\_a\_window\_update = (window\_update\_delta \geq \mathbf{int\_of\_num}(2 * cb.t\_maxseg) \vee$
$2 * window\_update\_delta \geq \mathbf{int\_of\_num}(sock.sf.n(\mathrm{SO\_RCVBUF})))$
in

(* Note that silly window avoidance and *max_sndwnd* need to be dealt with here; see `tcp_output.c:309` *)

(* Can a segment be transmitted? *)
let $do\_output = ($
(* Data to send and the send window has some space, or a FIN can be sent *)
$(have\_data\_or\_fin\_to\_send \wedge$
$(have\_data\_to\_send \implies snd\_wnd\_unused > 0)) \vee$ (* don't need space if only sending FIN *)

(* Can send a window update *)
$need\_to\_send\_a\_window\_update \vee$

(* There is outstanding urgent data to be transmitted *)
$\mathbf{is\_some}\ tcp\_sock.sndurp \vee$

(* An ACK should be sent immediately (e.g. in reply to a window probe) *)
$cb.tf\_shouldacknow$
$)$ **in**

let $persist\_fun =$
let $cant\_send = (\neg do\_output \wedge tcp\_sock.sndq \neq [\,] \wedge \mathrm{mode\_of}\ \ cb.tt\_rexmt = *)$ **in**
let $window\_shrunk = (win = 0 \wedge snd\_wnd\_unused < 0 \wedge$ (* $win = 0$ if in SYN_SENT, but still may send FIN *)
$(\mathrm{bsd\_arch}\ arch \implies tcp\_sock.st \neq \mathrm{SYN\_SENT}))$ **in**

if $cant\_send$ **then** (* takes priority over window_shrunk; note this needs to be checked *)
(* Can not transmit a segment despite a non-empty send queue and no running persist or retransmit timer. Must be the case that the receiver's advertised window is now zero, so start the persist timer. Normal: `tcp_output.c:378ff` *)
$\uparrow\lambda cb.cb\ \llparenthesis\ tt\_rexmt := \mathrm{start\_tt\_persist}\ \ 0\ cb.t\_rttinf\ \ arch \rrparenthesis$
else if $window\_shrunk$ **then**
(* The receiver's advertised window is zero and the receiver has retracted window space that it had previously advertised. Reset *snd_nxt* to *snd_una* because the data from *snd_una* to *snd_nxt* has likely not been buffered by the receiver and should be retransmitted. Bizzarely (on FreeBSD 4.6-RELEASE), if the persist timer is running reset its shift value *)
(* Window shrunk: |tcp_output.c:250ff| *)
$\uparrow\lambda cb.$
$cb\ \llparenthesis\ tt\_rexmt := \mathbf{case}\ \ cb.tt\_rexmt\ \ \mathbf{of}$
$\uparrow(((\textsc{Persist}, shift))_d) \to \uparrow(((\textsc{Persist}, 0))_d)$
$\|\ \_593 \to \mathrm{start\_tt\_persist}\ \ 0\ cb.t\_rttinf\ \ arch;$

$snd\_nxt := cb.snd\_una \rrbracket$
else
(* Otherwise, leave the persist timer alone *)
$*$
in
$(do\_output, persist\_fun)$

Description

This function determines if it is currently necessary to emit a segment. It is not quite a predicate, because in certain circumstances the operation of testing may start or reset the persist timer, and alter *snd_nxt*. Thus it returns a pair of a flag *do_output* (with the obvious meaning), and an optional mutator function *persist_fun* which, if present, performs the required updates on the TCP control block.

– **do TCP output :**
tcp_output_really $arch\ window\_probe\ ts\_val'\ ifds_0\ sock(sock', outsegs') =$
let $tcp\_sock =$ tcp_sock_of $sock$ **in**
let $cb = tcp\_sock.cb$ **in**

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

(* Note this does not deal with `TF_LASTIDLE` and `PRU_MORETOCOME` *)
let $snd\_cwnd' =$
if $\neg(cb.snd\_max = cb.snd\_una \wedge$
stopwatch_val_of $cb.t\_idletime \geq$ computed_rxtcur $cb.t\_rttinf\ arch)$
then (* inverted so this clause is tried first *)
$cb.snd\_cwnd$
else
(* The connection is idle and has been for >= 1RTO *)
(* Reduce *snd_cwnd* to commence slow start *)
$cb.t\_maxseg * ($**if** is_localnet $ifds_0($**the** $sock.is_2)$ **then** SS_FLTSZ_LOCAL **else** SS_FLTSZ) **in**

(* Calculate the amount of unused send window *)
let $win_0 =$ **min** $cb.snd\_wnd\ snd\_cwnd'$ **in**
let $win = ($**if** $window\_probe \wedge win_0 = 0$ **then** 1 **else** $win_0)$ **in**
let $(snd\_wnd\_unused : \mathsf{int}) =$ **int_of_num** $win - (cb.snd\_nxt - cb.snd\_una)$ **in**

(* Is it possible that a *FIN* may need to be transmitted? *)
let $fin_required = (sock.cantsndmore \wedge tcp_sock.st \notin \{$FIN_WAIT_2; TIME_WAIT$\})$ **in**

(* Calculate the sequence number after the last data byte in the send queue *)
let $last\_sndq\_data\_seq = cb.snd\_una +$ **length** $tcp\_sock.sndq$ **in**

(* The data to send in this segment (if any) *)
let $data' =$ DROP(**num**$(cb.snd\_nxt - cb.snd\_una))tcp\_sock.sndq$ **in**
let $data\_to\_send =$ TAKE(**min**(clip_int_to_num $snd\_wnd\_unused)cb.t\_maxseg)data'$ **in**

(* Should *FIN* be set in this segment? *)
let $FIN = (fin\_required \wedge cb.snd\_nxt +$ **length** $data\_to\_send \geq last\_sndq\_data\_seq)$ **in**

(* Should *ACK* be set in this segment? Under BSD, it is not set if the socket is in SYN_SENT and emitting a *FIN* segment due to shutdown() having been called. *)
let $ACK =$ **if** (bsd_arch $arch \wedge FIN \wedge tcp\_sock.st =$ SYN_SENT) **then** **F** **else** **T** **in**

(* If this socket has previously sent a *FIN* which has not yet been acked, and *snd_nxt* is past the *FIN*'s sequence number, then *snd_nxt* should be set to the sequence number of the *FIN* flag, i.e. a retransmission. Check that $snd\_una \neq iss$ as in this case no data has yet been sent over the socket *)

$$
\begin{array}{l}
\textbf{let } snd\_nxt' = \textbf{if } FIN \wedge (cb.snd\_nxt + \textbf{length } data\_to\_send = last\_sndq\_data\_seq + 1 \wedge \\
\qquad cb.snd\_una \neq cb.iss \vee \textbf{num}(cb.snd\_nxt - cb.iss) = 2)\ \textbf{then} \\
\qquad cb.snd\_nxt - 1 \\
\quad \textbf{else} \\
\qquad cb.snd\_nxt\ \textbf{ in}
\end{array}
$$

(* The BSD way: set *PSH* whenever sending the last byte of data in the send queue *)

$$\textbf{let } PSH = (data\_to\_send \neq [\,] \wedge cb.snd\_nxt + \textbf{length } data\_to\_send = last\_sndq\_data\_seq)\ \textbf{in}$$

(* If sending urgent data, set the *URG* and *urp* fields appropriately *)

$$
\begin{array}{l}
\textbf{let } (URG, urp) = (\textbf{case } tcp\_sock.sndurp \textbf{ of} \\
\quad * \rightarrow (\mathbf{F}, 0) \parallel \text{(* No urgent data; don't set *)} \\
\quad \uparrow sndurpn \rightarrow \textbf{let } urp_n = (cb.snd\_una + sndurpn) - cb.snd\_nxt + 1\ \textbf{ in} \\
\qquad\qquad \text{(* points one byte *past* the urgent byte *)} \\
\qquad \textbf{if } urp_n < 1\ \textbf{ then} \\
\qquad\quad (\mathbf{F}, 0)\ \text{(* Urgent data out of range; don't set *)} \\
\qquad \textbf{else if } urp_n < 65536\ \textbf{ then} \\
\qquad\quad (\mathbf{T}, \textbf{num } urp_n)\ \text{(* Urgent data in range; set *)} \\
\qquad \textbf{else} \\
\qquad\quad \text{(* Urgent data in the very distant future; set *)} \\
\qquad\quad \text{(* Steven's suggestion; not sure if followed *)} \\
\qquad\quad (\mathbf{T}, 65535))\ \textbf{in}
\end{array}
$$

(* Calculate size of the receive window (based upon available buffer space) *)

$$
\begin{array}{l}
\textbf{let } rcv\_wnd'' = \text{calculate\_bsd\_rcv\_wnd } sock.sf\ tcp\_sock\ \textbf{ in} \\
\textbf{let } rcv\_wnd' = \textbf{max}(\textbf{num}(cb.rcv\_adv - cb.rcv\_nxt))(\textbf{min}(\text{TCP\_MAXWIN} \ll cb.rcv\_scale) \\
\quad (\textbf{if } rcv\_wnd'' < sock.sf.n(\text{SO\_RCVBUF})\ \textbf{div } 4 \wedge rcv\_wnd'' < cb.t\_maxseg \\
\qquad \textbf{then } 0\ \text{(* Silly window avoidance: shouldn't advertise a tiny window *)} \\
\qquad \textbf{else } rcv\_wnd''))\ \textbf{in}
\end{array}
$$

(* Possibly set the segment's timestamp option. Under BSD, we may need to send a *FIN* segment from SYN_SENT, if the user called shutdown(), in which case the timestamp option hasn't yet been negotiated, so we used *tf_req_tstmp* rather than *tf_doing_tstmp*. *)

$$
\begin{array}{l}
\textbf{let } want\_tstmp = \textbf{if } (\text{bsd\_arch } arch \wedge tcp\_sock.st = \text{SYN\_SENT})\ \textbf{then } cb.tf\_req\_tstmp \\
\qquad\qquad \textbf{else } cb.tf\_doing\_tstmp\ \textbf{ in} \\
\textbf{let } ts = \text{do\_tcp\_options } want\_tstmp\ cb.ts\_recent\ ts\_val'\ \textbf{ in}
\end{array}
$$

(* Advertise an appropriately scaled receive window *)
(* Assert the advertised window is within a sensible range *)

$$
\begin{array}{l}
\textbf{let } win = \textbf{n2w}(rcv\_wnd' \gg cb.rcv\_scale)\ \textbf{in} \\
\textbf{w2n } win = rcv\_wnd' \gg cb.rcv\_scale \wedge
\end{array}
$$

(* Assert the urgent pointer is within a sensible range *)

$$
\begin{array}{l}
\textbf{let } urp\_ = \textbf{n2w } urp\ \textbf{ in} \\
\textbf{w2n } urp\_ = urp \wedge
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } seg = \langle\!|\ is_1 := sock.is_1; \\
\qquad is_2 := sock.is_2; \\
\qquad ps_1 := sock.ps_1; \\
\qquad ps_2 := sock.ps_2; \\
\qquad seq := snd\_nxt'; \\
\qquad ack := cb.rcv\_nxt; \\
\qquad URG := URG; \\
\qquad ACK := ACK; \\
\qquad PSH := PSH;
\end{array}
$$

$RST := \mathbf{F}$;
$SYN := \mathbf{F}$;
$FIN := FIN$;
$win := win$;
$ws := *$;
$urp := urp\_$;
$mss := *$;
$ts := ts$;
$data := data\_to\_send$
$\rrbracket$ **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**

(* Updated values to store in the control block after the segment is output *)
let $snd\_nxt'' = snd\_nxt' +$ **length** $data\_to\_send +$ (**if** FIN **then** 1 **else** 0) **in**
let $snd\_max' =$ **max** $cb.snd\_max\ snd\_nxt''$ **in**

(* Following a `tcp_output` code walkthrough by SB: *)
let $tt\_rexmt' =$ **if** (mode_of $cb.tt\_rexmt = * \lor$
(mode_of $cb.tt_rexmt = \uparrow($PERSIST$) \land \neg window_probe)) \land$
$snd\_nxt'' > cb.snd\_una$ **then**
(* If the retransmit timer is not running, or the persist timer is running and this segment isn't a window probe, and this segment contains data or a *FIN* that occurs past *snd_una* (i.e. new data), then start the retransmit timer. Note: if the persist timer is running it will be implicitly stopped *)
start_tt_rexmt $arch$ 0 $\mathbf{F}$ $cb.t\_rttinf$
else **if** ($window\_probe \lor$ (**is_some** $tcp\_sock.sndurp$)) $\land win_0 \neq 0 \land$
mode_of $cb.tt_rexmt = \uparrow($PERSIST$)$ **then**
(* If the segment is a window probe or urgent data is being sent, and in either case the send window is not closed, stop any running persist timer. Note: if *window_probe* is **T** then a persist timer will always be running but this isn't necessarily true when urgent data is being sent *)
$*$ (* stop persisting *)
else
(* Otherwise, leave the timers alone *)
$cb.tt\_rexmt$ **in**

(* Time this segment if it is sensible to do so, i.e. the following conditions hold : (a) a segment is not already being timed, and (b) data or a FIN are being sent, and (c) the segment being emitted is not a retransmit, and (d) the segment is not a window probe *)
let $t\_rttseg' =$ **if** $IS\_NONE\ cb.t\_rttseg \land (data\_to\_send \neq [\,] \lor FIN) \land$
$snd\_nxt'' > cb.snd\_max \land \neg window\_probe$
then
$\uparrow(ts\_val', snd\_nxt')$
else
$cb.t\_rttseg$ **in**

(* Update the socket *)

$$
\begin{aligned}
&sock' = sock \ \langle\!| \ pr := \mathrm{TCP\_PROTO}(tcp\_sock \\
&\qquad \langle\!| \ st := st'; cb := tcp\_sock.cb \\
&\qquad\quad \langle\!| \ tt\_rexmt := tt\_rexmt'; \\
&\qquad\qquad snd\_cwnd := snd\_cwnd'; \\
&\qquad\qquad rcv\_wnd := rcv\_wnd'; \\
&\qquad\qquad tf\_rxwin0sent := (rcv\_wnd' = 0); \\
&\qquad\qquad tf\_shouldacknow := \mathbf{F}; \\
&\qquad\qquad t\_rttseg := t\_rttseg'; \\
&\qquad\qquad snd\_max := snd\_max'; \\
&\qquad\qquad snd\_nxt := snd\_nxt''; \\
&\qquad\qquad tt\_delack := *; \\
&\qquad\qquad last\_ack\_sent := cb.rcv\_nxt; \\
&\qquad\qquad rcv\_adv := cb.rcv\_nxt + rcv\_wnd' \\
&\qquad\quad |\!\rangle |\!\rangle) |\!\rangle \wedge
\end{aligned}
$$

(* Constrain the list of output segments to contain just the segment being emitted *)

$outsegs' = [\mathrm{TCP}\ seg]$

Description

This function constructs the next segment to be output. It is usually called once tcp_output_required has returned true, but sometimes is called directly when we wish always to emit a segment. A large number of TCP state variables are modified also.

Note that while constructing the segment a variety of errors such as ENOBUFS are possible, but this is not modelled here. Also, window shrinking is not dealt with properly here.

– **combination of** tcp_output_required **and** tcp_output_really **:**

$$
\begin{aligned}
&\mathrm{tcp\_output\_perhaps}\ \ arch\ ts\_val\ ifds_0\ sock(sock', outsegs) = \\
&\mathbf{let}\ (do\_output, persist\_fun) = \mathrm{tcp\_output\_required}\ \ arch\ ifds_0\ sock\ \ \mathbf{in} \\
&\mathbf{let}\ \ sock'' = \\
&\mathbf{option\_case}\ sock\ (\lambda f.sock\ \langle\!|\ pr := \mathrm{TCP\_PROTO}(\mathrm{tcp\_sock\_of}\ \ sock\ \ cb\ :\hat{=}\ f)|\!\rangle)\ persist\_fun\ \mathbf{in} \\
&\mathbf{if}\ \ do\_output\ \ \mathbf{then} \\
&\ \mathrm{tcp\_output\_really}\ \ arch\ \mathbf{F}\ ts\_val\ ifds_0\ sock''(sock', outsegs) \\
&\ \mathbf{else} \\
&(sock' = sock'' \wedge outsegs = [\,])
\end{aligned}
$$

14.3 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.

14.3.1 Summary

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
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

mlift_tcp_output_perhaps_or_fail do mliftc for function returning at most one segment and not dealing with queueing flag

14.3.2 Rules

– **Attempt to enqueue segments, reverting appropriate socket fields if the enqueue fails :**
rollback_tcp_output $rcvdsyn\ seg\ arch\ rttab\ ifds\ is\_connect\ cb_0\ cb\_in(cb', es', outsegs') =$

(* 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 = test_outroute(seg, rttab, ifds, arch) in
let f0 = λcb.cb ⟪ (* revert to original values; on ip_output failure *)
                 snd_nxt := cb0.snd_nxt;
                 tt_delack := cb0.tt_delack;
                 last_ack_sent := cb0.last_ack_sent;
                 rcv_adv := cb0.rcv_adv
               ⟫ in
let f1 = λcb.if ¬rcvdsyn then
                 cb
               else
                 cb ⟪ (* set soft error flag; on ip_output routing failure *)
                     t_softerror := the route(* assumes route = SOME (SOME e) *)
                   ⟫ in
let f2 = λcb.cb ⟪ (* revert to original values; on early ENOBUFS *)
                 tf_rxwin0sent := cb0.tf_rxwin0sent;
                 t_rttseg := cb0.t_rttseg;
                 snd_max := cb0.snd_max;
                 tt_rexmt := cb0.tt_rexmt
               ⟫ in
let f3 = λcb.if is_some cb.tt_rexmt ∨ is_connect then (* quench; on ENOBUFS *)
                 cb
               else
                 cb ⟪ (* maybe start rexmt and close down window *)
                     tt_rexmt := start_tt_rexmt arch 0 F cb.t_rttinf;
                     snd_cwnd := cb.t_maxseg(* no LAN allowance, by design *)
                   ⟫ in
if ¬allocated then (* allocation failure *)
    cb' = f3(f2(f0 cb_in)) ∧ outsegs' = [] ∧ es' = ↑ ENOBUFS
 else if route = ∗ then (* ill-formed segment *)
    ASSERTION_FAILURE"rollback_tcp_output:1"(* should never happen *)
 else if ∃e.route = ↑(↑ e) then (* routing failure *)
    cb' = f1(f0 cb_in) ∧ outsegs' = [] ∧ es' = the route
 else if loopback_on_wire seg ifds then (* loopback not allowed on wire - RFC1122 *)
    (if windows_arch arch then
          cb' = cb_in ∧ outsegs' = [] ∧ es' = ∗(* Windows silently drops segment! *)
      else if bsd_arch arch then
          cb' = f0 cb_in ∧ outsegs' = [] ∧ es' = ↑ EADDRNOTAVAIL
      else if linux_arch arch then
          cb' = f0 cb_in ∧ outsegs' = [] ∧ es' = ↑ EINVAL
      else
          ASSERTION_FAILURE"rollback_tcp_output:2"(* never happen *)
    )
 else
    (∃queued.
```

$outsegs' = [(seg, queued)] \wedge$
if $\neg queued$ **then** (* queueing failure *)
$cb' = f3(f0\ cb\_in) \wedge es' = \uparrow$ 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**
$[\,] \to cb' = cb_0 \wedge oq' = oq$
$\|\ [seg] \to (\exists outsegs'\ es'.$
rollback_tcp_output $rcvdsyn\ seg\ arch\ rttab\ ifds$ **F** $cb_0\ cb\_in(cb', es', outsegs') \wedge$
enqueue_oq_list_qinfo$(oq, outsegs', oq'))$
$\|\ \_other84 \to ASSERTION\_FAILURE$"enqueue_or_fail"(* only 0 or 1 segments at a time *)
)

– **version of** enqueue_or_fail **that works with sockets rather than cbs :**
enqueue_or_fail_sock $rcvdsyn\ arch\ rttab\ ifds\ outsegs\ oq\ sock0\ sock(sock', oq') =$
(* NB: could calculate *rcvdsyn*, but clearer to pass it in *)
let $tcp\_sock =$ tcp_sock_of $sock$ **in**
let $tcp\_sock0 =$ tcp_sock_of $sock0$ **in**
$(\exists cb'.$
enqueue_or_fail $rcvdsyn\ arch\ rttab\ ifds\ outsegs\ oq($tcp_sock_of $sock0).cb($tcp_sock_of $sock).cb(cb', oq') \wedge$
$sock' = sock\ \llparenthesis\ pr :=$ 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 :**
mlift_tcp_output_perhaps_or_fail $ts\_val\ arch\ rttab\ ifds_0 =$
mliftc$(\lambda s(s', outsegs')$.
$\exists s_1\ segs.$
tcp_output_perhaps $arch\ ts\_val\ ifds_0\ s(s_1, segs) \wedge$
case $segs$ **of**

$[\,] \rightarrow s' = s_1 \wedge outsegs' = [\,]$
$\|\ [seg] \rightarrow (\exists cb'\ es'.$(* ignore error return *)
rollback_tcp_output $\mathbf{T}$ $seg\ arch\ rttab\ ifds_0$ $\mathbf{F}$
$(\text{tcp\_sock\_of}\ \ s).cb(\text{tcp\_sock\_of}\ \ s_1).cb(cb', es', outsegs') \wedge$
$s' = s_1\ \llparenthesis\ pr := \text{TCP\_PROTO}(\text{tcp\_sock\_of}\ \ s_1\ \llparenthesis\ cb := cb' \rrparenthesis)\rrparenthesis)$
$\|\ \_other58 \rightarrow ASSERTION\_FAILURE$"mlift_tcp_output_perhaps_or_fail"(* never happen *)
)

14.4 Incoming Segment Functions (TCP only)

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

14.4.1 Summary

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

14.4.2 Rules

– **Do updates appropriate to receiving a new segment on a connection :**
update_idle tcp_sock =
let $t\_idletime'$ = stopwatch_zero **in** (* update 'time most recent packet received' field *)
let $tt\_keep' = ($**if** $\neg(tcp\_sock.st = \text{SYN\_RECEIVED} \wedge tcp\_sock.cb.tf\_needfin)$ **then**
(* reset keepalive timer to 2 hours. *)
$\uparrow((())_{\text{slow\_timer}}\ \text{TCPTV\_KEEP\_IDLE})$
else
$tcp\_sock.cb.tt\_keep)$ **in**
let $tt\_fin\_wait\_2' = ($**if** $tcp\_sock.st = \text{FIN\_WAIT\_2}$ **then**
$\uparrow((())_{\text{slow\_timer}}\ \text{TCPTV\_MAXIDLE})$
else
$tcp\_sock.cb.tt\_fin\_wait\_2)$ **in**
$(t\_idletime', tt\_keep', tt\_fin\_wait\_2')$

14.5 Drop Segment Functions (TCP only)

When an erroneous or unexpected segment arrives, it is usually dropped (i.e, ignored). However, the peer is usually informed immediately by means of a RST or ACK segment.

14.5.1 Summary

dropwithreset	emit a RST segment corresponding to the passed segment, unless that would be stupid.
mlift_dropafterack_or_fail	send immediate ACK to segment, but otherwise process it no further
dropwithreset_ignore_fail	do emit_segs_pred, for function returning at most one seg and not dealing with queueing flag

14.5.2 Rules

– emit a RST segment corresponding to the passed segment, unless that would be stupid. :
dropwithreset $seg\ ifds_0\ ticks\ reason\ bndlm\ bndlm'\ outsegs =$
(* Needs list of the host's interfaces, to verify that the incoming segment wasn't broadcast. Returns a list of segments. *)

if (* never RST a RST *)
$seg.RST \lor$
(* is segment a (link-layer?) broadcast or multicast? *)
$\mathbf{F} \lor$
(* is source or destination broadcast or multicast? *)
$(\exists i_1.seg.is_1 = \uparrow i_1 \land \text{is\_broadormulticast } \emptyset\ i_1) \lor$
$(\exists i_2.seg.is_2 = \uparrow i_2 \land \text{is\_broadormulticast } ifds_0\ i_2)$
(* BSD only checks incoming interface, but should have same effect as long as interfaces don't overlap *)
then
$outsegs = [\,] \land bndlm' = bndlm$
else
(**choose** seg' :: make_rst_segment_from_seg seg.
let $(emit, bndlm'') = \text{bandlim\_rst\_ok}(seg', ticks, reason, bndlm)$ **in** (* finally: check if band-limited *)
$bndlm' = bndlm'' \land$
$outsegs =$ **if** $emit$ **then** [TCP seg'] **else** [ ])

– send immediate ACK to segment, but otherwise process it no further :
mlift_dropafterack_or_fail $seg\ arch\ rttab\ ifds\ ticks(sock, bndlm)((sock', bndlm', outsegs'), continue) =$
(* $ifds$ is just in case we need to send a RST, to make sure we don't send it to a broadcast address. *)
let $tcp\_sock = \text{tcp\_sock\_of } sock$ **in**
$(continue = \mathbf{T} \land$
let $cb = tcp\_sock.cb$ **in**
if $tcp\_sock.st = \text{SYN\_RECEIVED} \land$
$seg.ACK \land$
(**let** $ack = tcp\_seq\_flip\_sense\ seg.ack$ **in**
$(ack < cb.snd\_una \lor cb.snd\_max < ack))$
then
(* break loop in "LAND" DoS attack, and also prevent ACK storm between two listening ports that have been sent forged SYN segments, each with the source address of the other. (`tcp_input.c:2141`) *)
$sock' = sock \land$
dropwithreset $seg\ ifds\ ticks$ BANDLIM_RST_OPENPORT $bndlm\ bndlm'$(**map fst** $outsegs'$)
(* ignore queue full error *)
else
$(\exists sock_1\ msg\ cb'\ es'.$(* ignore errors *)
let $tcp\_sock1 = \text{tcp\_sock\_of } sock_1$ **in**
tcp_output_really $arch\ \mathbf{F}\ ticks\ ifds\ sock(sock_1, [msg]) \land$ (* did set tf_acknow and call tcp_output_perhaps, which seemed a bit silly *)
(* notice we here bake in the assumption that the timestamps use the same counter as the band limiter; perhaps this is unwise *)
rollback_tcp_output $\mathbf{T}\ msg\ arch\ rttab\ ifds\ \mathbf{F}\ tcp\_sock.cb\ tcp\_sock1.cb(cb', es', outsegs') \land$
$sock' = sock_1 \llparenthesis pr := \text{TCP\_PROTO}(tcp\_sock1 \llparenthesis cb := cb' \rrparenthesis) \rrparenthesis \land$
$bndlm' = bndlm))$

– do emit_segs_pred**, for function returning at most one seg and not dealing with queueing flag :**
dropwithreset_ignore_fail $seg\_in\ arch\ ifds\ rttab\ ticks\ reason\ b\ b'(outsegs' : (\mathsf{msg\#bool})\mathsf{list}) =$

(* No rollback necessary here. *)
$\exists segs.$
dropwithreset $seg\_in\ ifds\ ticks\ reason\ b\ b'\ segs \land$
case $segs$ **of**
$[\,] \to outsegs' = [\,]$
$\|\ [seg] \to$ (**choose** $allocated :: $ **if** INFINITE_RESOURCES **then** $\{\mathbf{T}\}$ **else** $\{\mathbf{T}; \mathbf{F}\}$.
if $\neg allocated$ **then**
$outsegs' = [\,]$
else
(**case** test_outroute$(seg, rttab, ifds, arch)$ **of**
$* \to ASSERTION\_FAILURE$"dropwithreset_ignore_fail:1"(* never happen *)
$\|\ \uparrow(\uparrow e) \to outsegs' = [\,]$(* ignore error *)
$\|\ \uparrow * \to \exists queued.outsegs' = [(seg, queued)]))$
$\|\ \_other57 \to ASSERTION\_FAILURE$"dropwithreset_ignore_fail:2"(* never happen *)

14.6 Close Functions (TCP only)

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

14.6.1 Summary

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

14.6.2 Rules

– **close the socket and remove the TCPCB :**
tcp_close $arch\ sock = sock$
$\langle\!|$ $cantrcvmore := \mathbf{T}$; (* 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** $sock.ps_1$;
$pr :=$ TCP_PROTO(tcp_sock_of $sock$
$\langle\!|$ $st :=$ CLOSED;
$cb :=$ initial_cb (* in reality, it's dropped entirely, but we don't do that *)
$\langle\!|$ $bsd\_cantconnect :=$ **if** bsd_arch $arch$ **then T else F**$|\!\rangle$;
$sndq := [\,]|\!\rangle)$
$|\!\rangle$

Description This is similar to BSD's `tcp_close()`, except that we do not actually remove the protocol/control blocks. The quad of the socket is cleared, to enable another socket to bind to the port we were previously using — this isn't actually done by BSD, but the effect is the same. The *bsd_cantconnect* flag is set to indicate that the socket is in such a detached state.

– **drop TCP connection, reporting the specified error. If synchronised, send RST to peer :**
tcp_drop_and_close $arch$ err $sock(sock', outsegs) =$
let $tcp\_sock =$ tcp_sock_of $sock$ **in** (
(**if** $tcp_sock.st \notin \{$CLOSED; LISTEN; SYN_SENT$\}$ **then**

$(\textbf{choose}\ \ seg :: (\text{make\_rst\_segment\_from\_cb}\ \ tcp\_sock.cb$
$\qquad\qquad\qquad(\textbf{the}\ \ sock.is_1, \textbf{the}\ \ sock.is_2, \textbf{the}\ \ sock.ps_1, \textbf{the}\ \ sock.ps_2)).$
$\quad outsegs = [\text{TCP}\ seg])$
$\ \ \textbf{else}$
$\quad outsegs = [\,]) \wedge$
$\textbf{let}\ \ es' =$
$\textbf{if}\ \ \mathsf{err} = \uparrow \text{ETIMEDOUT}\ \ \textbf{then}$
$\ (\textbf{if}\ \ tcp\_sock.cb.t\_softerror \neq *\ \ \textbf{then}$
$\quad tcp\_sock.cb.t\_softerror$
$\ \ \textbf{else}$
$\quad \uparrow \text{ETIMEDOUT})$
$\textbf{else}\ \ \textbf{if}\ \ \mathsf{err} \neq *\ \ \textbf{then}\ \ \mathsf{err}$
$\textbf{else}\ \ sock.es$
$\textbf{in}$
$sock' = \text{tcp\_close}\ \ arch(sock\ \llparenthesis\ es := es' \rrparenthesis))$

Description BSD calls this `tcp_drop`

Part XIII

TCP1_hostLTS

Chapter 15

Host LTS: Socket Calls

15.1 accept() (TCP only)

accept : fd → fd ∗ (ip ∗ port)

accept(fd) returns the next connection available on the completed connections queue for the listening TCP socket referenced by file descriptor fd. The returned file descriptor fd refers to the newly-connected socket; the returned ip and port are its remote address. accept() blocks if the completed connections queue is empty and the socket does not have the O_NONBLOCK flag set.

Any pending errors on the new connection are ignored, except for ECONNABORTED which causes accept() to fail with ECONNABORTED.

Calling accept() on a UDP socket fails: UDP is not a connection-oriented protocol.

15.1.1 Errors

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

EAGAIN	The socket has the O_NONBLOCK flag set and no connections are available on the completed connections queue.
ECONNABORTED	The connection at the head of the completed connections queue has been aborted; the socket has been shutdown for reading; or the socket has been closed.
EINVAL	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.

15.1.2 Common cases

accept() is called and immediately returns a connection: *accept_1*; *return_1*

accept() is called and blocks; a connection is completed and the call returns: *accept_2*; *deliver_in_99*; *deliver_in_1*; *accept_1*; *return_1*

15.1.3 API

Posix:	`int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`
FreeBSD:	`int accept(int s, struct sockaddr *addr, socklen_t *addrlen);`
Linux:	`int accept(int s, struct sockaddr *addr, socklen_t *addrlen);`
WinXP:	`SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen);`

In the Posix interface:

- `socket` is the listening socket's file descriptor, corresponding to the fd argument of the model;
- the returned `int` is either non-negative, i.e., a file descriptor referring to the newly-connected socket, or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `INVALID_SOCKET`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.
- `address` is a pointer to a `sockaddr` structure of length `address_len` corresponding to the ip $*$ port returned by the model accept(). If `address` is not a null pointer then it stores the address of the peer for the accepted connection. For the model accept() it will actually be a `sockaddr_in` structure; the peer IP address will be stored in the `sin_addr.s_addr` field, and the peer port will be stored in the `sin_port` field. If `address` is a null pointer then the peer address is ignored, but the model accept() always returns the peer address. On input the `address_len` is the length of the `address` structure, and on output it is the length of the stored address.

15.1.4 Model details

If the accept() call blocks then state ACCEPT2(sid) is entered, where sid is the index of the socket that accept() was called upon.

The following errors are not included in the model:

- `EFAULT` signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to accept() that is excluded by the clean interface used in the model.
- `EPERM` is a Linux-specific error code described by the Linux man page as "Firewall rules forbid connection". This is outside the scope of what is modelled.
- `EPROTO` is a Linux-specific error code described by the man page as "Protocol error". Only TCP and UDP are modelled here; the only sockets that can exist in the model are bound to a known protocol.
- `WSAECONNRESET` is a WinXP-specific error code described in the MSDN page as "An incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call." This error has not been encountered in exhaustive testing.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

From the Linux man page: Linux accept() passes already-pending network errors on the new socket as an error code from accept. This behaviour differs from other BSD socket implementations. For reliable operation the application should detect the network errors defined for the protocol after accept and treat them like EAGAIN by retrying. In case of TCP/IP these are ENETDOWN, EPROTO, ENOPROTOOPT, EHOSTDOWN, ENONET, EHOSTUNREACH, EOPNOTSUPP, and ENETUNREACH.

This is currently not modelled, but will be looked at when the Linux semantics are investigated.

15.1.5 Summary

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

15.1.6 Rules

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

$$\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d);\\
fds := fds;\\
files := files;\\
socks :=\\
socks\ \oplus\\
[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis,\\
\qquad\qquad\qquad\qquad [\,], *, [\,], *, \text{NO_OOBDATA})));\\
(sid', \textsc{Sock}(*, sf', \uparrow i_1', \uparrow p_1, \uparrow i_2, \uparrow p_2, es', cantsndmore', cantrcvmore',\\
\qquad\qquad \text{TCP_Sock}(\text{ESTABLISHED}, cb', *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{lbl}\ h\ \langle\!|ts := ts \oplus (tid \mapsto \big(\textsc{Ret}(\text{OK}(fd', (i_2, p_2)))\big)_{\text{sched_timer}});\\
fds := fds';\\
files := files \oplus [(fid', \textsc{File}(\text{FT_Socket}(sid'), \text{ff_default}))];\\
socks :=\\
socks\ \oplus\\
[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,\\
\qquad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis',\\
\qquad\qquad\qquad [\,], *, [\,], *, \text{NO_OOBDATA})));\\
(sid', \textsc{Sock}(\uparrow fid', sf', \uparrow i_1', \uparrow p_1, \uparrow i_2, \uparrow p_2, es',\\
\qquad cantsndmore', cantrcvmore', \text{TCP_Sock}(\text{ESTABLISHED}, cb', *, sndq,\\
\qquad\qquad\qquad sndurp, rcvq, rcvurp, iobc)))]|\!\rangle
\end{array}$$

$$\left(\left(\begin{array}{l}
t = \textsc{Run}\ \wedge\\
lbl = tid\cdot(\text{accept } fd)\ \wedge\\
rc = \text{FAST SUCCEED}\ \wedge\\
fid = fds[fd]\ \wedge\\
fd\ \in \mathbf{dom}(fds)\ \wedge\\
files[fid] = \textsc{File}(\text{FT_Socket}(sid), ff)
\end{array}\right) \vee \left(\begin{array}{l}
t = \textsc{Accept2}(sid)\ \wedge\\
\quad lbl = \tau\ \wedge\\
\quad rc = \text{SLOW urgent SUCCEED}
\end{array}\right)\right) \wedge$$

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

$(\forall i_1 . \uparrow i_1 = is_1 \implies i_1 = i_1')$

Description

This rule covers two cases: (1) the completed connection queue is non-empty when accept(fd) is called from a thread tid in the RUN state, where fd refers to a TCP socket sid, and (2) a previous call to accept(fd) on socket sid blocked, leaving its calling thread tid in state ACCEPT2(sid), and a new connection has become available.

In either case the listening TCP socket sid has a connection sid' at the head of its completed connections queue $sid' :: q$. A socket entry for sid' already exists in the host's finite map of sockets, $socks \oplus \ldots$. 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 (p**??**), 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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle \xrightarrow{tid\cdot(\text{accept}\ fd)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{ACCEPT2}(sid))_{\text{never\_timer}})|\!\rangle$$

$$\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
ff.b(\text{O_NONBLOCK}) = \mathbf{F} \wedge \\
(\exists sf\ is_1\ p_1\ cb\ lis\ es. \\
\quad h.socks[sid] = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, \mathbf{F}, cantrcvmore, \\
\qquad\qquad\qquad \text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \text{NO_OOBDATA})) \wedge \\
\quad lis.q = [\,])
\end{array}$$

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: $ff.b(\text{O\_NONBLOCK}) = \mathbf{F}$. 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 <u>**tcp: fast fail**</u> **Fail with EAGAIN: no pending connections and non-blocking semantics set**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d)\rangle\!\rangle \xrightarrow{tid\cdot(\mathsf{accept}\ fd)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL\ EAGAIN}))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$h.fds[fd] = fid \wedge$
$h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge$
$ff.b(\mathrm{O\_NONBLOCK}) = \mathbf{T} \wedge$
$(\exists sf\ is_1\ p_1\ cb\ lis\ es.$
$h.socks[sid] = \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,$
$\quad \mathrm{TCP\_Sock}(\mathrm{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \mathrm{NO\_OOBDATA})) \wedge$
$lis.q = [\,])$

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: $ff.b(\mathrm{O\_NONBLOCK}) = \mathbf{T}$. 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*·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 <u>**tcp: rc**</u> **Fail with ECONNABORTED: the listening socket has *cantsndmore* set or has become CLOSED. Returns either immediately or from a blocked state.**

$$\begin{array}{l}
h\ \langle\!\langle ts := ts \oplus (tid \mapsto (t)_d);\\
socks :=\\
socks \oplus\\
{[}(sid, \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,\\
\quad \mathrm{TCP_Sock}(st, cb, \uparrow lis, [\,], *, [\,], *, \mathrm{NO_OOBDATA})))]\rangle\!\rangle\\
\xrightarrow{lbl}\ h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL\ ECONNABORTED}))_{\mathrm{sched_timer}});\\
socks :=\\
socks \oplus\\
{[}(sid, \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,\\
\quad \mathrm{TCP_Sock}(st, cb, \uparrow lis, [\,], *, [\,], *, \mathrm{NO_OOBDATA})))]\rangle\!\rangle
\end{array}$$

$$\left(\begin{pmatrix} t = \mathrm{RUN} \wedge\\ st = \mathrm{LISTEN} \wedge\\ cantsndmore = \mathbf{T} \wedge\\ lbl = tid\cdot\mathsf{accept}(fd) \wedge\\ rc = \textsc{fast fail} \wedge\\ fd \in \mathbf{dom}(h.fds) \wedge\\ fid = h.fds[fd] \wedge\\ h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \end{pmatrix} \vee \begin{pmatrix} t = \mathrm{ACCEPT2}(sid) \wedge\\ ((cantrcvmore = \mathbf{T} \wedge st = \mathrm{LISTEN}) \vee\\ (st = \mathrm{CLOSED})) \wedge\\ lbl = \tau \wedge\\ rc = \textsc{slow}\ \mathrm{urgent}\ \textsc{fail} \end{pmatrix}\right)$$

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.

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 $\textsc{Accept2}(sid)$, it proceeds via an τ transition, otherwise by a $tid \cdot \mathsf{accept}(fd)$ transition. The thread is left in state $\textsc{Ret}(\text{FAIL ECONNABORTED})$ 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**

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

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

Description

It is not valid to call accept() on a socket that is not in the LISTEN state.

This rule covers two cases: (1) on the non-listening TCP socket *sid*, accept() is called from a thread *tid*, which is in the Run state, and (2) a previous call to accept() on TCP socket *sid* blocked because no completed connections were available, leaving thread *tid* in state $\textsc{Accept2}(sid)$ and after the accept() call blocked the socket changed to a state other than LISTEN.

In the first case the $\mathsf{accept}(fd)$ call on socket *sid*, referenced by file descriptor *fd*, proceeds by a $tid \cdot \mathsf{accept}(fd)$ transition and in the latter by a τ transition. In either case, the thread is left in state $\textsc{Ret}(\text{FAIL EINVAL})$ 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**

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

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

Description

This rule covers two cases: (1) from thread tid, which is in the RUN state, an accept(fd) call is made where fd refers to a TCP socket sid, and (2) a previous call to accept() blocked leaving thread tid in the ACCEPT2(sid) state. In either case the accept() call fails with EMFILE as the process (see Model Details) already has open its maximum number of open file descriptors OPEN_MAX.

In the first case the error is returned immediately (FAST FAIL) by performing an $tid \cdot$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**

$$h \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid \cdot \mathsf{accept}(fd)} h \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched\_timer}})\rangle\!\rangle$$

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

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 $tid \cdot$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.

15.2 bind() (TCP and UDP)

bind : (fd ∗ ip option ∗ port option) → unit

bind(fd, is, ps) assigns a local address to the socket referenced by file descriptor fd. The local address, (is, ps), may consist of an IP address, a port or both an IP address and port.

If bind() is called without specifying a port, bind$(_, _, *)$, the socket's local port assignment is autobound, i.e. an unused port for the socket's protocol in the host's ephemeral port range is selected and assigned to the socket. Otherwise the port p specified in the bind call, bind$(_, _, \uparrow p)$ 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 (p85).

A socket can only be bound once: it is not possible to rebind it to a different port later. A bind() call is not necessary for every socket: sockets may be autobound to an ephemeral port when a call requiring a port binding is made, e.g. connect().

15.2.1 Errors

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

EACCES	The specified port is in the privileged port range of the host architecture and the current thread does not have the required privileges to bind to it.
EADDRINUSE	The specified address is in use by or conflicts with the address of another socket using the same protocol. The error may occur in the following situations only: • 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.

15.2.2 Common cases

A server application creates a TCP socket and binds it to its local address. It is then put in the LISTEN state to accept incoming connections to this address: *socket_1*; *return_1*; *bind_1*; *return_1*; *listen_1*

A UDP socket is created and bound to its local address. recv() is called and the socket blocks, waiting to receive datagrams sent to the local address: *socket_1*; *return_1*; *bind_1*; *return_1*; *recv_12*

15.2.3 API

Posix:	`int bind(int socket, const struct sockaddr *address,` ` socklen_t address_len);`
FreeBSD:	`int bind(int s, struct sockaddr *addr, socklen_t addrlen);`
Linux:	`int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);`
WinXP:	`SOCKET bind(SOCKET s, const struct sockaddr* name, int namelen);`

In the Posix interface:

- `socket` is the socket's file descriptor, corresponding to the `fd` argument of the model.
- `address` is a pointer to a `sockaddr` structure of size `socklen_t` containing the local IP address and port to be assigned to the socket, corresponding to the *is* and *ps* arguments of the model. For the `AF_INET` sockets used in the model, a `sockaddr_in` structure stores the address. The `sin_addr.s_addr` field holds the IP address; if it is set to `0` then the IP address is wildcarded: $is = *$. The `sin_port` field stores the port to bind to; if it is set to `0` then the port is wildcarded: $ps = *$. On WinXP a wildcard IP is specified by the constant `INADDR_ANY`, not `0`
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo some argument renaming, except where noted above.

On Windows Socket 2 the `name` parameter is not necessarily interpreted as a pointer to a `sockaddr` structure but is cast this way for 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.

15.2.4 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.

15.2.5 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.

15.2.6 Rules

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

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

$$\begin{aligned}
&h_0 = h' \llparenthesis\ ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\\
&\qquad socks := socks \oplus\\
&\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, pr))]\\
&\quad \rrparenthesis \wedge\\
&h = h' \llparenthesis\ ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched_timer}});\\
&\qquad socks := socks \oplus\\
&\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, pr))];\\
&\qquad bound := bound \rrparenthesis \wedge\\
&fd \in \mathbf{dom}(h_0.fds) \wedge\\
&fid = h_0.fds[fd] \wedge\\
&h_0.files[fid] = \textsc{File}(\mathrm{FT_}\textsc{Socket}(sid), ff) \wedge\\
&sid \notin (\mathbf{dom}(socks)) \wedge\\
&(\forall i_1.is_1 = \uparrow i_1 \implies i_1 \in \mathrm{local_ips}(h_0.ifds)) \wedge\\
&p_1 \in \mathrm{autobind}(ps_1, (\mathrm{proto_of}\ pr), socks) \wedge\\
&bound = sid :: h_0.bound \wedge\\
&(h_0.privs \vee p_1 \notin \mathrm{privileged_ports}) \wedge\\
&\mathrm{bound_port_allowed}\ pr(h_0.socks \backslash\backslash sid) sf\ h_0.arch\ is_1\ p_1 \wedge\\
&(\mathbf{case}\ pr\ \mathbf{of}\\
&\quad \mathrm{TCP_PROTO}(tcp_sock) \to tcp_sock = \mathrm{TCP_Sock0}(\mathrm{CLOSED}, cb, *, [\,], *, [\,], *, \mathrm{NO_OOBDATA}) \wedge\\
&\qquad\qquad (\mathrm{bsd_arch}\ h_0.arch \implies cantsndmore = \mathbf{F} \wedge cb.bsd_cantconnect = \mathbf{F}) \parallel\\
&\quad \mathrm{UDP_PROTO}(udp_sock) \to udp_sock = \mathrm{UDP_Sock0}([\,]))
\end{aligned}$$

Description

The call $\mathsf{bind}(fd, is_1, ps_1)$ 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 $\mathsf{bind}()$ 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, $i_1 \in \mathrm{local\_ips}(h_0.ifds)$.

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 $\mathsf{bind}()$ 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 $\mathsf{bind}()$ 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 (p85) for further information.

The call proceeds by performing a $tid \cdot \mathsf{bind}(fd, is_1, ps_1)$ 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 <u>**all: fast fail**</u> **Fail with EADDRINUSE: the specified address is already in use**

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

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

Description

From thread *tid*, which is in the Run state, a $\mathsf{bind}(fd, is_1, \uparrow p_1)$ 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 (p85) for details.

The call proceeds by a $tid \cdot \mathsf{bind}(fd, is_1, \uparrow p_1)$ transition leaving the thread in state Ret(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**

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

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

Description

From thread *tid*, which is in the Run state, a $\mathsf{bind}(fd, \uparrow i_1, ps_1)$ call is made where *fd* refers to a socket *sid*.

The IP address, i_1, to be assigned as part of the socket's local address does not belong to any of the interfaces on the host, $i_1 \notin \text{local\_ips}(h.ifds)$, and therefore can not be assigned to the socket.

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

bind_5 **all: fast fail** **Fail with EINVAL: the socket is already bound to an address and does not support rebinding; or socket has been shutdown for writing on FreeBSD**

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

$fd \ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
$h.socks[sid] = sock \wedge$
$(sock.ps_1 \neq * \vee$
$(\text{bsd\_arch } h.arch \wedge sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$(sock.cantsndmore \vee$
$tcp\_sock.cb.bsd\_cantconnect)))$

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

Variations

FreeBSD	This rule also applies if *fd* refers to a TCP socket which is either shut down for writing or has its *bsd_cantconnect* flag set.

bind_7 **all: fast fail** **Fail with EACCES: the specified port is priveleged and the current process does not have permission to bind to it**

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

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

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

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

$fd \ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$

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

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(FAIL ENOBUFS) to return the out of resources error ENOBUFS to the caller.

Model details

Posix reports ENOBUFS to signify that "Insufficient resources were available to complete the call". This is not modelled here.

Variations

WinXP	On WinXP this error can occur non-deterministically when insufficient buffers are available.

15.3 close() (TCP and UDP)

close : fd → 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.

15.3.1 Errors

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

EAGAIN	The linger timer expired for a lingering 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.

15.3.2 Common cases

A TCP socket is created and connected to a peer; other socket calls are made, most likely send() and recv(), but the SO_LINGER option is not set. close() is then called and the connection is gracefully closed: *socket_1*; ...; *close_2*

A UDP socket is created and socket calls are made on it, mostly send() and recv() calls; the socket is then closed: *socket_1*; ...; *close_10*

15.3.3 API

Posix:	`int close(int fildes);`
FreeBSD:	`int close(int d);`
Linux:	`int close(int fd);`
WinXP:	`int closesocket(SOCKET s);`

In the Posix interface:

- `fildes` is the file descriptor to close, corresponding to the fd argument of the model close().
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.3.4 Model details

The following errors are not modelled:

- In Posix and on FreeBSD and Linux, `EIO` means an I/O error occurred while reading from or writing to the file system. Since we model only sockets, not file systems, we do not model this error.
- On FreeBSD, `ENOSPC` means the underlying object did not fit, cached data was lost.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.3.5 Summary

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

15.3.6 Rules

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

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

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

Description

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

The $\mathsf{close}(fd)$ call proceeds by a $tid \cdot \mathsf{close}(fd)$ transition leaving the host in the successful return state $\textsc{Ret}(\mathrm{OK}())$. 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 $\mathsf{close}()$ call does not alter the socket's state as other file descriptors still refer to the socket through file description fid.

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

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

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

$(st \in \{\mathrm{ESTABLISHED}; \mathrm{FIN\_WAIT\_1}; \mathrm{CLOSING}; \mathrm{FIN\_WAIT\_2};$
$\quad \mathrm{TIME\_WAIT}; \mathrm{CLOSE\_WAIT}; \mathrm{LAST\_ACK}\} \vee$

$st = \mathrm{SYN\_RECEIVED} \wedge \mathrm{linux\_arch}\ \ h.arch) \wedge$
$(sf.t(\mathrm{SO\_LINGER}) = \infty \vee$
$ff.b(\mathrm{O\_NONBLOCK}) = \mathbf{T} \wedge sf.t(\mathrm{SO\_LINGER}) \neq 0 \wedge \neg\, \mathrm{linux\_arch}\ \ h.arch) \wedge$
$fd \ \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$\mathrm{fid\_ref\_count}(fds, fid) = 1 \wedge$
$fds' = fds \backslash\backslash fd \wedge$
$fid \ \notin (\mathbf{dom}(files))$

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: $\mathrm{fid\_ref\_count}(fds, fid) = 1$. 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, $sf.t(\mathrm{SO\_LINGER}) = \infty$, and regardless of whether the socket is in non-blocking mode or not, i.e. $ff.b(\mathrm{O\_NONBLOCK})$ is unconstrained, the call to close() proceeds successfully without blocking.

On all platforms except for Linux, if the socket is in non-blocking mode $ff.b(\mathrm{O\_NONBLOCK}) = \mathbf{T}$ the linger option may be set with a positive duration: $sf.t(\mathrm{SO\_LINGER}) \neq 0$). 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 $\mathrm{R{\small ET}}(\mathrm{OK}())$. 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, $(sid, \mathrm{S{\small OCK}}(\uparrow fid, \ldots,))$ 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, $ff.b(\mathrm{O\_NONBLOCK}) = \mathbf{T}$ and a linger option is set to a non-zero value, $sf.t(\mathrm{SO\_LINGER}) \neq 0$, the socket may block on a call to close(). See also *close_4* (p140).

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

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{R{\small UN}})_d); fds := fds; files := files \oplus [(fid, \mathrm{F{\small ILE}}(\mathrm{FT\_S{\small OCKET}}(sid), ff))]; socks := socks \oplus [(sid, sock)]; oq := oq|\!\rangle$$
$$\xrightarrow{tid\cdot \mathsf{close}(fd)}$$
$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{R{\small ET}}(\mathrm{OK}()))_{\mathrm{sched\_timer}}); fds := fds'; files := files; socks := socks \oplus [(sid, sock')]; oq := oq'|\!\rangle$$

$(st \ \in \{\mathrm{ESTABLISHED}; \mathrm{FIN\_WAIT\_1}; \mathrm{CLOSING}; \mathrm{FIN\_WAIT\_2};$
$\quad \mathrm{TIME\_WAIT}; \mathrm{CLOSE\_WAIT}; \mathrm{LAST\_ACK}\} \vee$
$st = \mathrm{SYN\_RECEIVED} \wedge \mathrm{linux\_arch}\ \ h.arch) \wedge$
$sock = \mathrm{S{\small OCK}}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$sf.t(\mathrm{SO\_LINGER}) = 0 \wedge$
$fd \ \in \mathbf{dom}(fds) \wedge$

$fid = fds[fd] \wedge$
$\mathrm{fid\_ref\_count}(fds, fid) = 1 \wedge$
$fds' = fds \backslash\backslash fd \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin \mathbf{dom}(socks) \wedge$
$sock' = (\mathrm{tcp\_close}\ h.arch\ sock)\langle\!|\ fid := *|\!\rangle \wedge$
$seg \in \mathrm{make\_rst\_segment\_from\_cb}\ cb(i_1, i_2, p_1, p_2) \wedge$
$\mathrm{enqueue\_and\_ignore\_fail}\ h.arch\ h.rttab\ h.ifds[\mathrm{TCP}\ seg]oq\ oq'$

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: $\mathrm{fid\_ref\_count}(fds, fid) = 1$. 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, $sf.t(\mathrm{SO\_LINGER}) = 0$, to signify that an abortive closure of socket *sid* is required.

The close(fd) call proceeds by a $tid \cdot \mathrm{close}(fd)$ transition leaving the host in the successful return state $\mathrm{Ret}(\mathrm{OK}())$. 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 (p118), 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 (p121). Additionally, its file description field is cleared.

Variations

Linux	The socket can be in the SYN_RECEIVED state or in one of the synchronised states $\geq$ ESTABLISHED.

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

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

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

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

Description

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

On Linux, the socket is also permitted to be in the SYN_RECEIVED state and it may have non-blocking semantics $\mathit{ff}.b(\mathrm{O\_NONBLOCK}) = \mathbf{T}$, 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), $\mathit{sf}.t(\mathrm{SO\_LINGER}) \notin \{0; \infty\}$. 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 $\mathsf{close}(\mathit{fd})$ call proceeds by a $\mathit{tid} \cdot \mathsf{close}(\mathit{fd})$ transition leaving the host in the blocked state $\mathrm{C{\scriptsize LOSE}2}(\mathit{sid})$. The socket is marked as unable to send and receive further data, $\mathit{cantsndmore} = \mathbf{T} \land \mathit{cantrcvmore} = \mathbf{T}$; 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, $(\mathit{sid}, \mathrm{S{\scriptsize OCK}}(\uparrow \mathit{fid}, \ldots))$, 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 $\geq$ ESTABLISHED. On Linux, non-blocking semantics do not take precedence over the SO_LINGER option, i.e. if the socket is non-blocking, $\mathit{ff}.b(\mathrm{O\_NONBLOCK}) = \mathbf{T}$ and a linger option is set to a non-zero value, $\mathit{sf}.t(\mathrm{SO\_LINGER}) \neq 0$ the socket may block on a call to $\mathsf{close}()$.

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

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{C{\scriptsize LOSE}2}(sid))_d);\\
socks := socks \oplus\\
\quad [(sid, \mathrm{S{\scriptsize OCK}}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},\\
\qquad \mathrm{TCP\_Sock}(st, cb, *, [\,], sndurp, [\,], rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{\tau}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize ET}}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\\
socks := socks \oplus\\
\quad [(sid, \mathrm{S{\scriptsize OCK}}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},\\
\qquad \mathrm{TCP\_Sock}(st, cb, *, [\,], sndurp, [\,], rcvurp, iobc)))]|\!\rangle
\end{array}
$$

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

Description

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

The rule proceeds via a τ transition leaving thread *tid* in the $\textsc{Ret}(\mathrm{OK}())$ 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**

$$
\begin{array}{lcl}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Close2}(sid))_d); & \xrightarrow{\tau} & h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{FAIL\ EAGAIN}))_{\mathrm{sched\_timer}});\\
socks := socks \oplus [(sid, sock)] & & socks := socks \oplus [(sid, sock)]\\
|\!\rangle & & |\!\rangle
\end{array}
$$

$$
\begin{array}{l}
sock = \textsc{Sock}(*, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \mathbf{T}, \mathbf{T},\\
\qquad\qquad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, [\,], rcvurp, iobc)) \wedge\\
\mathrm{timer\_expires}\ \ d \wedge\\
st\ \notin \{\mathrm{TIME\_WAIT}; \mathrm{CLOSED}\}
\end{array}
$$

Description

A previous call to close() with the linger option set on the socket blocked, leaving thread *tid* in the $\textsc{Close2}(sid)$ state. The linger timer has expired, timer_expires d, before the socket has been successfully closed: $st\ \notin \{\mathrm{TIME\_WAIT}; \mathrm{CLOSED}\}$.

The rule proceeds via a τ transition leaving thread *tid* in the $\textsc{Ret}(\mathrm{FAIL\ EAGAIN})$ 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. $st\ \notin \{\mathrm{TIME\_WAIT}; \mathrm{CLOSED}\}$. 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.**

$$
\begin{array}{l}
\quad h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\\
\quad fds := fds;\\
\quad files := files \oplus [(fid, \textsc{File}(\mathrm{FT\_}\textsc{Socket}(sid), ff))];\\
\quad socks := socks \oplus [(sid, sock)]|\!\rangle\\
\xrightarrow{tid \cdot \mathsf{close}(fd)} \quad h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\\
\qquad\qquad\quad fds := fds';\\
\qquad\qquad\quad files := files;\\
\qquad\qquad\quad socks := socks|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
(tcp\_sock.st\ \in \{\mathrm{CLOSED}; \mathrm{SYN\_SENT}\} \vee\\
tcp\_sock.st = \mathrm{SYN\_RECEIVED} \wedge \neg\, \mathrm{linux\_arch}\ \ h.arch) \wedge\\
\mathrm{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge\\
fid\ \notin (\mathbf{dom}(files)) \wedge\\
sid\ \notin (\mathbf{dom}(socks)) \wedge\\
fd\ \in \mathbf{dom}(fds) \wedge\\
fid = fds[fd] \wedge\\
\mathrm{fid\_ref\_count}(fds, fid) = 1 \wedge
\end{array}
$$

$fds' = fds \backslash\backslash fd$

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: $st \in \{\text{CLOSED}; \text{SYN\_SENT}\}$.

The close(fd) call proceeds by a tid·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**

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\\
fds := fds;\\
files := files \oplus [(fid, \text{FILE}(\text{FT\_SOCKET}(sid), ff))];\\
socks := socks \oplus [(sid, sock)];\\
\text{listen} := \text{listen};\\
oq := oq|\!\rangle\\
\xrightarrow{tid\cdot\text{close}(fd)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}});\\
\quad fds := fds';\\
\quad files := files;\\
\quad socks := socks';\\
\quad \text{listen} := listen';\\
\quad oq := oq'|\!\rangle
\end{array}
$$

$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$fd \in \mathbf{dom}(fds) \wedge$
$fid = fds[fd] \wedge$
$\text{fid\_ref\_count}(fds, fid) = 1 \wedge$
$fid \notin (\mathbf{dom}(files)) \wedge$
$sid \notin (\mathbf{dom}(socks)) \wedge$

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

(* BSD and Linux do not send RSTs to sockets on $lis.q_0$. *)
$socks\_to\_rst = \{(sock', tcp\_sock') \mid \exists sid'.sid' \in lis.q \wedge$

$\qquad sock' = socks[sid'] \wedge$
$\qquad \text{TCP\_PROTO}(tcp\_sock') = sock'.pr \wedge$

$tcp\_sock'.st \notin \{\text{CLOSED}; \text{LISTEN}; \text{SYN\_SENT}\}\} \wedge$

$socks\_to\_rst\_list \in \text{ORDERINGS } socks\_to\_rst \wedge$

$\textbf{card } socks\_to\_rst = \textbf{length } segs \wedge$

$(\textbf{let } make\_rst\_seg = \lambda(sock', tcp\_sock').$
$\quad \text{make\_rst\_segment\_from\_cb } tcp\_sock'.cb(\textbf{the } sock'.is_1, \textbf{the } sock'.is_2, \textbf{the } sock'.ps_1, \textbf{the } sock'.ps_2)$
$\textbf{in}$

$\textbf{every I}(\textbf{map2}(\lambda s'\ seg'.seg' \in make\_rst\_seg\ s')socks\_to\_rst\_list\ segs)) \wedge$

(* Note this is a clear example of where fuzzy timing is needed: should these really all have exactly the same time always? *)
$\text{enqueue\_each\_and\_ignore\_fail } h.arch\ h.rttab\ h.ifds(\textbf{map} \text{ TCP } segs)oq\ oq' \wedge$

$fds' = fds \backslash\backslash fd \wedge$
$listen' = \textbf{filter}(\lambda sid'.sid' \neq sid)\text{listen} \wedge$
$socks' = socks|_{\{sid' | sid' \notin lis.q_0 @ lis.q\}}$

Description

A close(fd) call is performed on the TCP socket *sock* referenced by file descriptor fd which is the only file descriptor referencing the socket's file description fid, fid_ref_count$(fds, fid) = 1$. Socket *sock* is locally bound to port p_1 and one or more local IP addresses is_1, and is in the LISTEN state.

The listening socket *sock* may have ESTABLISHED incoming connections on its connection queue $lis.q$ and incomplete incoming connection attempts on queue $lis.q_0$. Each connection, regardless of whether it is complete or not, is represented by a socket entry in $h.socks$ and its corresponding index *sid* is on the respective queue. These connections have not been accepted by any thread through a call to accept() and are dropped on the closure of socket *sock*.

A set of reset seqments *rsts_to_go* is created using the auxiliary function make_rst_segment_from_cb (p109) for each of the sockets referenced by both queues. This is performed by looking up each socket $sock'$ for every sid' in the concatentation of both queues, $lis.q_0$ @ $lis.q$, and extracting their address quads $(sock'.is_1, sock'.is_2, sock'.ps_1, sock'.ps_2)$ and control blocks *cb* for use by make_rst_segment_from_cb.

The set of reset segments *rsts_to_go* is constrained to a list, *segs*, and queued by the auxiliary function enqueue_each_and_ignore_fail on the hosts output queue $h.oq$. The enqueue_each_and_ignore_fail function always succeeds; if it is not possible to add any of the reset segments *segs* to the output queue $h.oq$, the corresponding error codes are ignored and the reset segments in error are ultimately not queued for transmission. This is sensible behaviour as the sockets for these connections are about to be deleted: if a reset segment does not successfully abort the remote end of the connection, perhaps because it could not be transmitted in the first place, any future incoming segments should not match any other socket in the system and will be dropped.

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

In the final host state, the mapping of file descriptor fd to file descriptor index fid is removed from the file descriptors finite map $fds' = fds \backslash\backslash fd$ and file description entry fid is removed from the finite map of file descriptors $h.files$. The socket entry *sock* is removed from the hosts finite map of sockets $h.socks$ and the socket's *sid* value is removed from the host's list of listening sockets h.listen by $listen' = \textbf{filter}(\lambda sid'.sid' \neq sid)\text{listen}$. Finally, all the sockets in $h.socks$ that were referenced on one of the queues $lis.q_0$ and $lis.q$, are removed by $socks' = socks|_{\{sid' | sid' \notin lis.q_0 @ lis.q\}}$ as they were not accepted by any thread before socket *sock* was closed.

Model details

The local IP address option is_1 of the socket *sock* is not constrained in this rule. Instead it is constrained by other rules for bind() and listen() prior to the socket entering the LISTEN state.

close_10 **udp: fast succeed** **Successfully close the last file descriptor of a UDP socket**

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

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

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*, $\mathrm{fid\_ref\_count}(fds, fid) = 1$. From thread *tid*, which is in the RUN state, a close(*fd*) call is made and succeeds.

A *tid*·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 $fds' = fds \backslash\backslash fd$.

15.4 connect() (TCP and UDP)

connect : fd ∗ ip ∗ port option → 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 $\mathsf{connect}(\mathsf{fd}, \mathsf{ip}, *)$ will fail with an EADDRNOTAVAIL error; on Linux and WinXP such a call succeeds: datagrams from any port on the host with IP address ip will be delivered to the socket. Calling connect() on a UDP socket that already has a peer address set is allowed: the peer address will be replaced with the one specified in the call. On FreeBSD if the socket has a pending error, that may be returned when the call is made, and the peer address will also be set.

In order for a socket to connect to a peer or have its peer address set, it must be bound to a local IP and port. If it is not bound to a local port when the connect() call is made, then it will be autobound: an unused port for the socket's protocol in the host's ephemeral port range is selected and assigned to the socket. If the socket does not have its local IP address set then it will be bound to the primary IP address of an interface which has a route to the peer. If the socket does have a local IP address set then the interface that this IP address will be the one used to connect to the peer; if this interface does not have a route to the peer then for a TCP socket the connect() call will fail when the SYN is enqueued on the host's outqueue; for a UDP socket the call will fail on FreeBSD, whereas on Linux and WinXP the connect() call will succeed but later send() calls to the peer will fail.

For a TCP socket, its binding quad must be unique: there can be no other socket in the host's finite map of sockets with the same binding quad. If the connect() call would result in two sockets having the same binding quad then it will fail with an EADDRINUSE error. For UDP sockets the same is true on FreeBSD, but on Linux and WinXP multiple sockets may have the same address quad. The socket that matching datagrams are delivered to is architecture-dependent: see *lookup* (p**??**).

15.4.1 Errors

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

EADDRNOTAVAIL	There is no route to the peer; a port must be specified ($\mathsf{port} \neq *$); 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.

15.4.2 Common cases

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

15.4.3 API

Posix:	`int connect(int socket, const struct sockaddr *address, socklen_t address_len);`
FreeBSD:	`int connect(int s, const struct sockaddr *name, socklen_t namelen);`
Linux:	`int connect(int sockfd, 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.

15.4.4 Model details

If the call blocks then the thread enters state CONNECT2(sid) where sid is the identifier of the socket attempting to establish a connection.

The following errors are not modelled:

- `EAFNOSUPPORT` means that the specified address is not a valid address for the address family of the specified socket. The model connect() only models the `AF_INET` family of addresses so this error cannot occur.
- `EFAULT` signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to connect() that is excluded by the clean interface used in the model.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `EINVAL` is a Posix-specific error signifying that the `address_len` argument is not a valid length for the socket's address family or invalid address family in the `sockaddr` structure. The length of the address to connect to is implicit in the model connect(), and only the `AF_INET` family of addresses is modelled so this error cannot occur.
- `EPROTOTYPE` is a Posix-specific error meaning that the specified address has a different type than the socket bound to the specified peer address. This error does not occur in any of the implementations as TCP and UDP sockets are dealt with seperately.
- `EACCES`, `ELOOP`, and `ENAMETOOLONG` are errors dealing with Unix domain sockets which are not modelled here.

15.4.5 Summary

connect_1	**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

15.4.6 Rules

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

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

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

$$
\begin{aligned}
h = h_0 \,\langle\!|\; & ts := ts\_ \oplus (tid \mapsto (\mathrm{RUN})_d);\\
& socks := socks \oplus\\
& \quad [(sid, \mathrm{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
& \qquad \mathrm{TCP\_Sock}(st, cb, *, [\,], *, [\,], *, \mathrm{NO\_OOBDATA})))];\\
& oq := oq \;|\!\rangle \wedge
\end{aligned}
$$

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

$$
\begin{aligned}
h' = h_0 \,\langle\!|\; & ts := ts\_ \oplus (tid \mapsto t');\\
& socks := socks \oplus\\
& \quad [(sid, \mathrm{SOCK}(\uparrow fid, sf, \uparrow i_1', \uparrow p_1', is_2', ps_2', es'', \mathbf{F}, \mathbf{F},\\
& \qquad \mathrm{TCP\_Sock}(st', cb''', *, [\,], *, [\,], *, \mathrm{NO\_OOBDATA})))];\\
& bound := bound;\\
& oq := oq' \;|\!\rangle \wedge
\end{aligned}
$$

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

$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge$

(* 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. *)

$i_1' \in \mathrm{auto\_outroute}(i_2, is_1, h.rttab, h.ifds) \wedge$
(* 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' *)
$p_1' \in \mathrm{autobind}(ps_1, \mathrm{PROTO\_TCP}, h.socks) \wedge$
(* 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*. *)

$$
\begin{aligned}
&((st = \mathrm{CLOSED} \wedge is_2 = * \wedge ps_2 = * \wedge\\
&\quad es = * \wedge cantsndmore = \mathbf{F} \wedge (cantrcvmore = \mathbf{F} \vee \mathrm{bsd\_arch}\ h.arch)) \vee\\
&(\mathrm{bsd\_arch}\ h.arch \wedge st = \mathrm{TIME\_WAIT} \wedge\\
&\quad (is_2 \neq * \implies ps_2 \neq *) \wedge\\
&\qquad (ps_2 \neq * \implies is_2 \neq *))) \wedge
\end{aligned}
$$

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

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

(* Pick an initial sequence number non-deterministically. This allows accidental spoofing of our own connections, but it is unclear how a tighter specification should be expressed. *)
$iss \in \{n \mid \mathbf{T}\} \wedge$

(* If windows-scaling is to be requested for the connection then $request\_r\_scale = \uparrow n$ where n is a valid window scale; otherwise, $request\_r\_scale = *$. rcv_wnd0 is a valid receive window size. If window scaling is to be requested then the socket's receive window is set to rcv_wnd0 scaled by the window scale factor n; otherwise it is set to rcv_wnd0. The socket's receive window is not greater than the size of the socket's receive buffer. We must allow implementations to either (a) not implement window scaling, or (b) choose on a per-connection basis whether to do window scaling or not. This permits both. *)

$(request\_r\_scale : \mathsf{num\ option}) \in \{*\} \cup \{\uparrow n \mid n \geq 0 \wedge n \leq \text{TCP\_MAXWINSCALE}\} \wedge$
$(rcv\_wnd0 : \mathsf{num}) \in \{n \mid n > 0 \wedge n \leq \text{TCP\_MAXWIN}\} \wedge$
$(rcv\_wnd : \mathsf{num}) = rcv\_wnd0 \ll (\textbf{option\_case}\ 0\ \mathbf{I}\ request\_r\_scale) \wedge$
$rcv\_wnd \leq sf.n(\text{SO\_RCVBUF}) \wedge$

(* Either advertise a maximum segment size, $advmss$, that is between 1 and $65535 - 40$, or advertise no maximum segment size. If one is advertised, $advmss' = \uparrow advmss$; otherwise, $advmss' = *$. *)

$advmss\ \in \{n \mid n \geq 1 \wedge n \leq (65535 - 40)\} \wedge$
$advmss'\ \in \{*; \uparrow advmss\} \wedge$

(* If time-stamping is to be requested for the connection, then $tf\_req\_tstmp' = \mathbf{T}$; otherwise $tf\_req\_tstmp' = \mathbf{F}$. *)

$tf\_req\_tstmp'\ \in \{\mathbf{F}; \mathbf{T}\} \wedge$ (* do timestamp? *)

(* If there is no segment currently being timed for this socket (the expected case) then the SYN segment will be timed, with t_rttseg' set to the current time and the initial sequence number for the connection, iss. *)

$(\textbf{let}\ t\_rttseg' = \textbf{if}\ IS\_NONE\ cb.t\_rttseg\ \textbf{then}$
$\qquad \uparrow(\text{ticks\_of}\ h.ticks, iss)$
$\quad \textbf{else}$
$\qquad cb.t\_rttseg\ \textbf{in}$

(* Update the socket's control block to cb', which is cb except we: (1) start the retransmit and connection establishment timers; (2) set the snd_una, snd_nxt, snd_max, iss fields based on the initial sequence number chosen; (3) set the rcv_wnd, rcv_adv, and $tf\_rxwin0sent$ fields based on the receive window chosen; (4) record whether or not to do windows scaling, time-stamping, and what the advertised maximum segment size is; and (5) store the segment to time. *)

$cb' = cb\ \langle\!|\ tt\_rexmt := \text{start\_tt\_rexmtsyn}\ h.arch\ 0\ \mathbf{F}\ cb.t\_rttinf;$
$\quad tt\_conn\_est := \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_INIT}});$

$\quad snd\_una := iss;$
$\quad snd\_nxt := iss + 1;$
$\quad snd\_max := iss + 1;$
$\quad iss := iss;$
$\quad rcv\_wnd := rcv\_wnd;$
$\quad rcv\_adv := cb.rcv\_nxt + rcv\_wnd;$
(* since rcv_nxt is 0 at this point (since we do not yet know), this is a bit odd. But it models BSD behaviour. *)

$\quad tf\_rxwin0sent := (rcv\_wnd = 0);$
$\quad request\_r\_scale := request\_r\_scale;$ (* store whether we requested WS and if so what *)
$\quad t\_maxseg := cb.t\_maxseg;$ (* do not change this *)
$\quad t_{advmss} := advmss';$ (* store what mss we advertised; $*$ or $\uparrow v$ *)
$\quad tf\_req\_tstmp := tf\_req\_tstmp';$
$\quad last\_ack\_sent := \text{tcp\_seq\_foreign}\ 0w;$
$\quad t\_rttseg := t\_rttseg'$

$\rrbracket) \wedge$

(* 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_segment (p106) for details. *)
choose seg :: make_syn_segment $cb'(i_1', i_2, p_1', p_2)$(ticks_of $h.ticks$).

(* and send it out... *)

(* If possible, enqueue the segment *seg* on the host's outqueue. The auxiliary function rollback_tcp_output (p117) is used for this; if the segment is a well-formed segment, there is a route to the peer from i_1', and there are no buffer allocation failures, $outsegs' \neq [\,]$, then the segment is enqueued on the host's outqueue, oq, resulting in a new outqueue, oq'. The socket's control block is left as cb' which is described above. Otherwise an error may have occurred; possible errors are: (1) ENOBUFS indicating a buffer allocation failure; (2) a routing error; or (3) EADDRNOTAVAIL on FreeBSD or EINVAL on Linux indicating that the segment would cause a loopback packet to appear on the wire (on WINXP the segment is silently dropped with no error in this case). If an error does occur then the socket's control block reverts to cb, the control block when the call was made. *)

$\exists outsegs'$.
rollback_tcp_output **F**(TCP seg)$h.arch$ $h.rttab$ $h.ifds$ **T**
$(cb \llparenthesis snd\_nxt := iss;$
$snd\_max := iss;$
$tt\_delack := *;$
$last\_ack\_sent := \mathrm{tcp\_seq\_foreign}\ 0w;$
$rcv\_adv := \mathrm{tcp\_seq\_foreign}\ 0w$
$\rrbracket) cb'(cb'', es', outsegs') \wedge$
$cb''' = (\textbf{if}\ (outsegs' \neq [\,] \vee \mathrm{windows\_arch}\ h.arch)\ \textbf{then}\ cb''\ \textbf{else}\ cb) \wedge$
enqueue_oq_list_qinfo$(oq, outsegs', oq') \wedge$

(* If the socket is a blocking one, its O_NONBLOCK flag is not set, then the call will block, entering state CONNECT2(sid) and leaving the socket in state SYN_SENT with peer address $(\uparrow i_2, \uparrow p_2)$ and, if the segment could not be enqueued, its pending error set to the error resulting from the attempt to enqueue the segment.
If the socket is non-blocking, its O_NONBLOCK flag is set, and the segment was enqueued on the host's outqueue, then the call will fail with an EINPROGRESS error (or EAGAIN on WinXP). The socket will be left in state SYN_SENT with peer address $(\uparrow i_2, \uparrow p_2)$. Otherwise, if the segment was not enqueued, then the call will fail with the error resulting from attempting to enqueue it, $\uparrow err$; the socket will be left in state CLOSED with no peer address set. *)

(* In the case of BSD, if we connect via the loopback interface, then the segment exchange occurs so fast that the socket has connected before the connect-calling thread regains control. When it does, it sees that the socket has been connected, and therefore returns with success rather than EINPROGRESS. Since this behaviour is due to timing, however, it may be possible for the connect call to return before all the segments have been sent, for example if there was an artificially imposed delay on the loopback interface. This behaviour is therefore made nondeterministic, for a BSD non-blocking socket connecting via loopback, in that it may either fail immediately, or be blocked for a short time. Linux does not exhibit this behaviour.*)

((* blocking socket, or BSD and using loopback interface *)
$((\neg ff.b(\mathrm{O\_NONBLOCK}) \vee (\mathrm{bsd\_arch}\ h.arch \wedge i_2 \in \mathrm{local\_ips}\ h.ifds)) \wedge$
$t' = (\mathrm{CONNECT2}(sid))_{\mathrm{never\_timer}} \wedge rc = \mathrm{BLOCK} \wedge$
$es'' = es' \wedge st' = \mathrm{SYN\_SENT} \wedge is_2' = \uparrow i_2 \wedge ps_2' = \uparrow p_2) \vee$
(* non-blocking socket *)
$(ff.b(\mathrm{O\_NONBLOCK}) \wedge$
$es = * \wedge$
$(err = (\textbf{if}\ \mathrm{windows\_arch}\ h.arch\ \textbf{then}\ \mathrm{EAGAIN}\ \textbf{else}\ \mathrm{EINPROGRESS}) \vee \uparrow err = es') \wedge$
$t' = (\mathrm{RET}(\mathrm{FAIL}\ err))_{\mathrm{sched\_timer}} \wedge rc = \mathrm{FAST\ FAIL} \wedge es'' = * \wedge$
if $oq = oq'$ **then**
$st' = \mathrm{CLOSED} \wedge is_2' = * \wedge ps_2' = *$
else

$st' = \text{SYN\_SENT} \land is'_2 = \uparrow i_2 \land ps'_2 = \uparrow p_2)$
)

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 $es' = \text{ENOBUFS}$; 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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{CONNECT2}\ sid)_d)|\!\rangle \xrightarrow{\tau} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}})|\!\rangle$$

$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \land$
$tcp\_sock.st \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \land$
$(\neg \exists tid'\ d'.(tid' \in \mathbf{dom}(ts)) \land (tid' \neq tid) \land$
$ts[tid'] = (\text{CONNECT2}\ sid)_{d'})$

Description

Thread *tid* is blocked in state CONNECT2(*sid*) where *sid* identifies a TCP socket which is in state ESTABLISHED: the connection establishment has been successfully completed; or CLOSE_WAIT: connection establishment successfully completed but a *FIN* was received during establishment. *tid* is the only thread which is blocked waiting for the socket *sid* to establish a connection. As connection establishment has now completed, the thread can successfully return from the blocked state.

A τ transition is made, leaving the thread state RET(OK()).

connect_3 **tcp: slow urgent fail** **Fail with the pending error on a socket in the CLOSED state**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{CONNECT2}\ sid)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle \xrightarrow{\tau} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ e))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle$$

$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$tcp\_sock.st = \text{CLOSED} \wedge$
$(\text{bsd\_arch}\ h.arch \implies tcp\_sock.cb.bsd\_cantconnect = \mathbf{T})$

Description

Thread *tid* is blocked in the CONNECT2(*sid*) state where *sid* identifies a TCP socket *sock* that is in the CLOSED state: connection establishment has failed, leaving the socket in a pending error state $\uparrow e$. Usually this occurs when there is no listening TCP socket at the peer address, giving an error of ECONNREFUSED or ECONNRESET; or when the connection establishment timer expired, giving an error of ETIMEDOUT. The call now returns, failing with the error e, and clearing the pending error field of the socket.

A τ transition is made, leaving the thread state RET(FAIL e).

Variations

FreeBSD	When connection establishment failed, the *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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{CONNECT2}\ sid)_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle \xrightarrow{\tau} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ err))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock')]|\!\rangle$$

$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, \uparrow err, \mathbf{F}, \mathbf{F},$
$\quad \text{TCP\_Sock}(\text{SYN\_SENT}, cb, *, [\,], *, [\,], *, \text{NO\_OOBDATA})) \wedge$
(* On WinXP if the error is from routing to an unavailable address, the error is not returned and the socket is left alone. The rexmtsyn timer will retry the SYN transmission and eventually fail. *)
$\neg(\text{windows\_arch}\ h.arch \wedge err = \text{EINVAL}) \wedge$
$(\textbf{if}\ \text{bsd\_arch}\ h.arch\ \textbf{then}$
$\quad (\textbf{if}\ (err = \text{EADDRNOTAVAIL})\ \textbf{then}$

$\qquad sock' = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, ps_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, \mathbf{F},$
$\qquad\quad \text{TCP\_Sock}(\text{SYN\_SENT}, cb, *, [\,], *, [\,], *, \text{NO\_OOBDATA}))$
$\quad \textbf{else}$
$\qquad sock' = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F},$
$\qquad\quad \text{TCP\_Sock}(\text{CLOSED}, \text{initial\_cb}, *, [\,], *, [\,], *, \text{NO\_OOBDATA})))$
$\textbf{else}$
(* close the socket, but do not shutdown for reading/writing *)
$\quad sock' = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, ps_1, *, *, *, \mathbf{F}, \mathbf{F},$
$\qquad \text{TCP\_Sock}(\text{CLOSED}, cb', *, [\,], *, [\,], *, \text{NO\_OOBDATA})) \wedge$
$\quad cb' = \text{initial\_cb}$
$)$

Description

Thread *tid* is blocked in the $\text{C}\scriptstyle{\text{ONNECT}}\textstyle 2(sid)$ 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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d); socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow err|\!\rangle)]|\!\rangle \xrightarrow{tid\cdot\mathsf{connect}(fd,\, i_2,\, \uparrow p_2)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ err))_{\text{sched\_timer}}); socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle$$

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

Description

From thread *tid*, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ 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 $tid\cdot\mathsf{connect}(fd, \mathsf{ip}, \mathsf{port})$ 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 $\mathsf{connect}(fd, \_, \_)$ call to have previously been made. The call fails, setting the pending error on the socket, and when connect() is called to check the status of connection establishment the error is returned. In such a case *err* is most likely to be ECONNREFUSED, ECONNRESET, or ETIMEDOUT.

connect_5 **tcp: fast fail** **Fail with EALREADY, EINVAL, EISCONN, EOPNOTSUPP: socket already in use**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle \xrightarrow{tid\cdot\mathsf{connect}(fd,\, i_2,\, \uparrow p_2)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ err))_{\text{sched\_timer}})|\!\rangle$$

$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff)\ \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr\ \wedge$
case $tcp\_sock.st$ **of**
  $\text{SYN\_SENT} \rightarrow$ **if** $ff.b(\text{O\_NONBLOCK}) = \mathbf{T}$ **then** $err = \text{EALREADY}$ (* connection already in progress *)

else if windows_arch $h.arch$ **then** err = EALREADY (* connection already in progress *)
else if bsd_arch $h.arch$ **then** err = EISCONN (* connection being established *)
else $ASSERTION\_FAILURE$"connect_5:1" ‖ (* never happen *)
SYN_RECEIVED → **if** $ff.b$(O_NONBLOCK) = **T** **then** err = EALREADY (* connection already in progress *)
else if windows_arch $h.arch$ **then** err = EALREADY
else if bsd_arch $h.arch$ **then** err = EISCONN (* connection being established *)
else $ASSERTION\_FAILURE$"connect_5:2" ‖ (* never happen *)
LISTEN → **if** windows_arch $h.arch$ **then** err = EINVAL (* socket is listening *)
else if bsd_arch $h.arch$ **then** err = EOPNOTSUPP
else if linux_arch $h.arch$ **then** err = EISCONN
else $ASSERTION\_FAILURE$"connect_5:3" ‖ (* never happen *)
ESTABLISHED → err = EISCONN ‖ (* socket already connected *)
FIN_WAIT_1 → err = EISCONN ‖ (* socket already connected *)
FIN_WAIT_2 → err = EISCONN ‖ (* socket already connected *)
CLOSING → err = EISCONN ‖ (* socket already connected *)
CLOSE_WAIT → err = EISCONN ‖ (* socket already connected *)
LAST_ACK → err = EISCONN ‖ (* socket already connected; seems that fd is valid in this state *)
TIME_WAIT → (windows_arch $h.arch$ ∨ linux_arch $h.arch$) ∧ err = EISCONN ‖
(* BSD allows a TIME_WAIT socket to be reconnected *)
CLOSED → err = EINVAL ∧ bsd_arch $h.arch$ ∧ $tcp\_sock.cb.bsd\_cantconnect$ = **T**

Description

From thread tid, which is in the RUN state, a 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·connect($fd, i_2, \uparrow p_2$) transition is made, leaving the thread state RET(FAIL err).

Variations

FreeBSD	If the socket is in state TIME_WAIT then the call does not fail: the socket may be reconnected by *connect_1* (p148).

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

h ⦇$ts := ts \oplus (tid \mapsto (\text{RUN})_d)$;
$socks := socks \oplus$
$[(sid, sock ⦇is_1 := *; ps_1 := ps_1⦈)]$⦈

$\xrightarrow{tid\cdot\text{connect}(fd, i_2, \uparrow p_2)}$ h ⦇$ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched\_timer}})$;
$socks := socks \oplus$
$[(sid, sock ⦇is_1 := is'_1; ps_1 := ps'_1⦈)]$;
$bound := bound$⦈

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
(**if** bsd_arch $h.arch$ ∧ proto_of $sock.pr$ = PROTO_TCP **then**
$is'_1 = \uparrow i'_1 \wedge i'_1 \in local\_primary\_ips\ h.ifds \wedge$
$ps'_1 = \uparrow p'_1 \wedge p'_1 \in \text{autobind}(ps_1, \text{PROTO\_TCP}, h.socks) \wedge$

$$
\begin{array}{l}
\quad (\textbf{if}\ \ ps_1 = * \ \ \textbf{then}\ \ bound = sid :: h.bound\ \ \textbf{else}\ \ bound = h.bound) \\
\ \textbf{else}\ \ is_1' = * \wedge ps_1' = ps_1 \wedge bound = h.bound) \wedge \\
\textbf{case}\ \ \text{test\_outroute\_ip}(i_2, h.rttab, h.ifds, h.arch)\ \textbf{of} \\
\quad \uparrow e \rightarrow err = e \\
\ \| \ \_other29 \rightarrow \mathbf{F} \wedge \\
(\text{proto\_of}\ \ sock.pr = \text{PROTO\_UDP} \implies \neg\, \text{bsd\_arch}\ \ h.arch)
\end{array}
$$

Description

From thread tid, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ call is made. fd refers to a socket identified by sid which does not have a local IP address set. The test_outroute_ip (p82) function is used to check if there is a route from the host to i_2. There is no route so the call will fail with a routing error err. If there is no interface with a route to the host then on Linux the call fails with ENETUNREACH and on FreeBSD and WinXP it fails with EHOSTUNREACH. If there are interfaces with a route to the host but none of these are up then the call fails with ENETDOWN.

A $tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)$ 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**

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

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

Description

From thread tid, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ 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 $tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state RET(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**

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

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

Description

From thread tid, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ is made. fd refers to a socket identified by sid which is not bound to a local port. There are no ephemeral ports available to autobind to so the call fails with an EADDRNOTAVAIL error.

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

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

$$h \, \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d) |\!\rangle \xrightarrow{tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)} h \, \langle\!| ts := ts \oplus (tid \mapsto (\text{CONNECT2}(sid))_{\text{never\_timer}}) |\!\rangle$$

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

Description

From thread tid, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ 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 $tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)$ 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**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d);\ socks := socks \oplus [(sid, sock\ \langle\!\langle cantsndmore := \mathbf{T}; pr := \text{TCP\_PROTO}(tcp\ \langle\!\langle st := \text{CLOSED}\rangle\!\rangle)\rangle\!\rangle)]\rangle\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET(FAIL EINVAL)})_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!\langle cantsndmore := \mathbf{T}; pr := \text{TCP\_PROTO}(tcp\ \langle\!\langle st := \text{CLOSED}\rangle\!\rangle)\rangle\!\rangle)]\rangle\!\rangle$$

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

Description

On FreeBSD, from thread tid, which is in the RUN state, a $\mathsf{connect}(fd, i_2, \uparrow p_2)$ 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 $tid \cdot \mathsf{connect}(fd, i_2, \uparrow p_2)$ transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

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

connect_7 **udp: fast succeed** **Set peer address on socket with binding quad** $*, ps_1, *, *$

$$h_0$$

$$\xrightarrow{tid \cdot \mathsf{connect}(fd, i_2, ps_2)}$$

h_0 $\langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1', \uparrow p_1', \uparrow i_2, ps_2, es, cantsndmore', cantrcvmore, \text{UDP\_PROTO}(udp)))];$
$bound := bound$
$|\!\rangle$

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

Description

Consider a UDP socket *sid*, referenced by *fd*, with no local IP or peer address set. From thread *tid*, which is in the RUN state, a $\text{connect}(fd, i_2, ps_2)$ call is made. The socket's local port is either set to p_1', or it is unset and can be autobound to a local ephemeral port p_1'. The local IP address can be set to i_1' which is the primary IP address for an interface with a route to i_2.

A $tid \cdot \text{connect}(fd, i_2, ps_2)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. The socket's local address is set to $(\uparrow i_1', \uparrow p_1')$, and its peer address is set to $(\uparrow i_2, ps_2)$. If the socket's local port was autobound then *sid* is placed at the head of the host's list of bound sockets: $bound = sid :: h_0.bound$.

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

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

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

$h_0 = h \; \langle\!| \; ts := ts \oplus (tid \mapsto (\text{RUN})_d);$

$socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \mathrm{UDP\_PROTO}(udp)))]\rangle\!\rangle \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\mathrm{FT\_S}\textsc{ocket}(sid), ff) \wedge$
$(\mathrm{bsd\_arch}\ h.arch \implies ps \neq * \wedge es = *) \wedge$
$(\mathbf{if}\ \mathrm{windows\_arch}\ h.arch\ \mathbf{then}\ cantsndmore' = \mathbf{F}$
$\mathbf{else}\ cantsndmore' = cantsndmore) \wedge$
$\neg(\exists(sid', s) :: (h_0.socks \backslash\backslash sid).$
$s.is_1 = \uparrow i_1 \wedge s.ps_1 = \uparrow p_1 \wedge$
$s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge$
$\mathrm{proto\_of}\ s.pr = \mathrm{PROTO\_UDP} \wedge$
$\mathrm{bsd\_arch}\ h.arch)$

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 $\mathsf{connect}(fd, i, ps)$ call is made.

The call succeeds: a $tid \cdot \mathsf{connect}(fd, i, ps)$ 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 $(\uparrow i_1', \uparrow p1', \uparrow i, ps)$.
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**

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

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

Description

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

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

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

$$\begin{aligned}
&\text{bsd_arch } h.arch \wedge \\
&h_0 = h \;\langle\!\!| \; ts := ts \oplus (tid \mapsto (\textsc{Run})_d); \\
&\qquad socks := socks \oplus \\
&\qquad\quad [(sid, sock \;\langle\!\!| \; es := \uparrow err; pr := \text{UDP_PROTO}(udp) |\!\!\rangle)] |\!\!\rangle \wedge \\
&fd \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&h.files[fid] = \textsc{File}(\text{FT_}\textsc{Socket}(sid), ff) \wedge \\
&ps \neq * \wedge \\
&\neg(\exists (sid', s) :: (h_0.socks \backslash\backslash sid). \\
&\qquad s.is_1 = sock.is_1 \wedge s.ps_1 = sock.ps_1 \wedge \\
&\qquad s.is_2 = \uparrow i \wedge s.ps_2 = ps \wedge \\
&\qquad \text{proto_of } s.pr = \text{PROTO_UDP})
\end{aligned}$$

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 $\mathsf{connect}(fd, i, ps)$ 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 $tid \cdot \mathsf{connect}(fd, i, ps)$ transition is made, leaving the thread state $\textsc{Ret}(\text{FAIL } err)$. The socket's peer address is set to $(\uparrow i, ps)$, and the error is cleared from the socket.

Variations

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

15.5 disconnect() (TCP and UDP)

disconnect : fd → 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* (p139)).

15.5.1 Errors

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

EADDRNOTAVAIL	There are no ephemeral ports left for autobinding to.
EAFNOSUPPORT	The address family `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.

15.5.2 Common cases

disconnect_1; *return_1*

15.5.3 API

disconnect() is a Posix connect() call with the address family set to `AF_UNSPEC`.

Posix: `int connect(int socket, const struct sockaddr *address, socklen_t address_len);`

FreeBSD: `int connect(int s, const struct sockaddr *name, socklen_t namelen);`

Linux: `int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);`

WinXP: `int connect(SOCKET s, const struct sockaddr* name, int namelen);`

In the Posix interface:

- `socket` is a file descriptor referring to a socket. This corresponds to the `fd` argument of the model disconnect().
- `address` is a pointer to a location of size `address_len` containing a `sockaddr` structure which specifies the address to connect to. For a disconnect() call, the `sin_family` field of the `sockaddr` family must be set to `AF_UNSPEC`; other fields can be set to anything.
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The Linux man-page states: "Unconnecting a socket by calling connect with a AF_UNSPEC address is not yet implemented." As a result, a disconnect() call always returns successfully on Linux.

The WinXP documentation states: "The default destination can be changed by simply calling `connect` again, even if the socket is already connected. Any datagrams queued for receipt are discarded if `name` is different from the previous `connect`." This implies that calling disconnect() will result in all datagrams on the socket's receive queue; however, this is not the case: no datagrams are discarded.

15.5.4 Summary

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

15.5.5 Rules

disconnect_4 **tcp: fast fail** **Fail with EAFNOSUPPORT: address family not supported; EOPNOTSUPP: operation not supported; EALREADY: connection already in progress; or EISCONN: socket already connected**

$$h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{R}\small{\text{UN}})_d)|\!\rangle \xrightarrow{tid\cdot \mathsf{disconnect}(fd)} h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{R}\small{\text{ET}}(\text{FAIL } err))_{\text{sched\_timer}})|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(\text{FT\_Socket}(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
$\neg(\text{linux\_arch } h.arch) \wedge$
case $tcp\_sock.st$ **of**
  CLOSED → **if** bsd_arch $h.arch$ **then**
    if $tcp\_sock.cb.bsd\_cantconnect = \mathbf{T}$ **then** err = EINVAL
    else 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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Run})_d);\ socks := socks \oplus [(sid, sock)];\ oq := oq|\!\rangle \xrightarrow{tid\cdot\text{disconnect}(fd)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Ret}(\text{OK}()))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock')];\ oq := oq'|\!\rangle$$

$\text{linux\_arch}\ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(\text{FT\_Socket}(sid), ff) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$(\textbf{if}\ tcp\_sock.st \in \{\text{SYN\_RECEIVED}; \text{ESTABLISHED}; \text{FIN\_WAIT\_1}; \text{FIN\_WAIT\_2}; \text{CLOSE\_WAIT}\}\ \textbf{then}$
$\quad \text{tcp\_drop\_and\_close}\ h.arch * sock(sock', outsegs) \wedge$
$\quad \text{enqueue\_and\_ignore\_fail}\ h.arch\ h.rttab\ h.ifds\ outsegs\ oq\ oq'$
$\ \textbf{else}$
$\quad sock = sock' \wedge$
$\quad oq = oq')$

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**

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

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

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

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

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.

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

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

h_0 $\langle\!\langle ts := ts\_ \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched\_timer}});$
$socks := socks \oplus$
$[(sid, \text{SOCK}(\uparrow fid, sf, *, \uparrow p_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))];$
$bound := sid :: h_0.bound\rangle\!\rangle$

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

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*·disconnect(*fd*) transition is made, leaving the thread either in state Ret(OK()), or in state Ret(FAIL EAFNOSUPPORT). The socket is autobound to a local ephemeral port $p1'$, and *sid* is placed on the head of the host's list of bound sockets.

Variations

FreeBSD	As above: the call fails with an EAFNOSUPPORT error.
Linux	As above: the call succeeds.
WinXP	As above: the call fails with an EAFNOSUPPORT error.

disconnect_3 **udp: fast fail** **Fail with EAGAIN, EADDRNOTAVAIL, or ENOBUFS: there are no ephemeral ports left**

$$h_0 \xrightarrow{tid\cdot \mathsf{disconnect}\ fd} h_0\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{sched\_timer}})|\!\rangle$$

$$\begin{aligned}
&h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\\
&\qquad socks := socks \oplus\\
&\qquad\quad [(sid, \mathrm{SOCK}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \mathrm{UDP_PROTO}(udp)))]|\!\rangle \wedge\\
&fd \in \mathbf{dom}(h.fds) \wedge\\
&fid = h.fds[fd] \wedge\\
&h.files[fid] = \mathrm{FILE}(\mathrm{FT_SOCKET}(sid), ff) \wedge\\
&\mathrm{autobind}(*, \mathrm{PROTO_UDP}, h_0.socks) = \emptyset \wedge\\
&e \in \{\mathrm{EAGAIN}; \mathrm{EADDRNOTAVAIL}; \mathrm{ENOBUFS}\}
\end{aligned}$$

Description

Consider a UDP socket *sid* referenced by *fd* and with binding quad $*, *, *, *$. From thread *tid*, which is in the Run state, a 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.

15.6 dup() (TCP and UDP)

dup : fd → 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.

15.6.1 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.

15.6.2 Common cases

dup_1; *return_1*

15.6.3 API

Posix:	`int dup(int fildes);`
FreeBSD:	`int dup(int oldd);`
Linux:	`int dup(int oldfd);`

In the Posix interface:

- `fildes` is a file descriptor referring to the open file description for which another file descriptor is to be created for. This corresponds to the fd argument of the model dup().
- The returned `int` is either non-negative to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. If the call is successful then the returned `int` is the new file descriptor corresponding to the fd return type of the model dup().

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

15.6.4 Summary

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

15.6.5 Rules

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

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize UN}})_d);\ fds := fds|\!\rangle \xrightarrow{tid \cdot \mathsf{dup}(fd)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize ET}}(\mathrm{OK}\ fd'))_{\mathrm{sched\_timer}});\ fds := fds'|\!\rangle$$

$$\begin{aligned}
&\mathrm{unix_arch}\ \ h.arch \wedge \\
&fd\ \in \mathbf{dom}(h.fds) \wedge \\
&fid = h.fds[fd] \wedge \\
&\mathrm{nextfd}\ \ h.arch\ fds\ fd' \wedge \\
&fd' < \mathrm{OPEN_MAX_FD} \wedge \\
&fds' = fds \oplus (fd', fid)
\end{aligned}$$

Description

From thread *tid*, which is in the RUN state, a dup(*fd*) call is made where *fd* is a file descriptor referring to an open file description identified by *fid*. A new file descriptor, fd' can be created in an architecture-specific way according to the nextfd (p**??**) function. fd' is less than the maximum open file descriptor, OPEN_MAX_FD. The call succeeds returning fd'.

A $tid \cdot$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**

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

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

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.

A *tid*·dup(*fd*) transition is made, leaving the thread state RET(FAIL EMFILE).

Variations

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

15.7 dupfd() (TCP and UDP)

dupfd : fd ∗ int → 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.

15.7.1 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.

15.7.2 Common cases

dupfd_1; *return_1*

15.7.3 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.

15.7.4 Model details

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

15.7.5 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

15.7.6 Rules

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

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\ fds := fds|\!\rangle \xrightarrow{tid\cdot \mathsf{dupfd}(fd,n)} h\ \langle\!|ts := ts \oplus (tid \mapsto \big(\mathrm{RET}(\mathrm{OK}\ fd')\big)_{\mathrm{sched\_timer}});\ fds := fds'|\!\rangle$$

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

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**

$$h \ \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d) |\!\rangle \xrightarrow{tid \cdot \mathsf{dupfd}(fd, n)} h \ \langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } err))_{\text{sched\_timer}}) |\!\rangle$$

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

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

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

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

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.

15.8 getfileflags() (TCP and UDP)

getfileflags : fd → filebflag list

A call to getfileflags(fd) returns a list of the file flags currently set for the file which fd refers to.
The possible file flags are:

- O_ASYNC Reports whether signal driven I/O is enabled.

- O_NONBLOCK Reports whether a socket is non-blocking.

15.8.1 Errors

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

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

15.8.2 Common cases

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

15.8.3 API

getfileflags() is Posix `fcntl(fd,F_GETFL)`. On WinXP it is `ioctlsocket()` with the `FIONBIO` command.

Posix:	`int fcntl(int fildes, int cmd, ...);`
FreeBSD:	`int fcntl(int fd, int cmd, ...);`
Linux:	`int fcntl(int fd, int cmd);`
WinXP:	`int ioctlsocket(SOCKET s, long cmd, u_long* argp)`

In the Posix interface:

- `fildes` is a file descriptor for the file to retrieve flags from. It corresponds to the `fd` argument of the model getfileflags(). On WinXP the `s` is a socket descriptor corresponding to the `fd` argument of the model getfileflags().
- `cmd` is a command to perform an operation on the file. This is set to `F_GETFL` for the model getfileflags(). On WinXP, `cmd` is set to `FIONBIO` to get the O_NONBLOCK flag; there is no O_ASYNC flag on WinXP.
- The call takes a variable number of arguments. For the model getfileflags() only the two arguments described above are needed.
- If the call succeeds the returned `int` represents the file flags that are set corresponding to the filebflag list return type of the model getfileflags(). If the returned `int` is `-1` then an error has occurred in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR` with the actual error code available through a call to `WSAGetLastError()`.

15.8.4 Model details

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `WSAENOTSOCK` is a possible error on WinXP as the `ioctlsocket()` call is specific to a socket. In the model the getfileflags() call is performed on a file.

15.8.5 Summary

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

15.8.6 Rules

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

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d)\rangle\!\rangle \quad \xrightarrow{tid\cdot \mathsf{getfileflags}(fd)} \quad h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{OK}\ flags))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

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

Description

From thread *tid*, which is in the RUN state, a getfileflags(fd) call is made. fd refers to a file description FILE(ft, ff) where ff is the file flags that are set. The call succeeds, returning *flags* which is a list representing some ordering of the boolean file flags $ff.b$ in ff.

A $tid\cdot$getfileflags(fd) transition is made, leaving the thread state RET(OK($flags$)).

15.9 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().

15.9.1 Errors

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

15.9.2 Common cases

getifaddrs_1; *return_1*

15.9.3 API

getifaddrs() is two calls to Posix `ioctl()`: one with the `SIOCGIFCONF` request and one with the `SIOCGIFNETMASK` request. On FreeBSD there is a specific `getifaddrs()` call. On WinXP the getifaddrs() call does not exist.

Posix:	`int ioctl(int fildes, int request, ...  /* arg */);`
FreeBSD:	`int getifaddrs(struct ifaddrs **ifap);`
Linux:	`int ioctl(int d, int request, ...);`

In the Posix interface:

- `fildes` is a file descriptor. There is no corresponding argument in the model getifaddrs().
- `request` is the operation to perform on the file. When `request` is `SIOCGIFCONF` the list of all interfaces is returned; when it is `SIOCNETMASK` the subnet mask is returned for an interface.
- The function takes a variable number of arguments. When `request` is `SIOCGIFCONF` there is a third argument: a pointer to a location to store a linked-list of the interfaces; when it is `SIOCGIFNETMASK` it is a pointer to a structure containing the interface and it is filled in with the subnet mask for that interface.
- The returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`.

To construct the return value of type $(ifid * \mathsf{ip} * \mathsf{ip\ list} * netmask)\mathsf{list}$, the interface name and the IP addresses associated with it are obtained from the call to `ioctl()` using `SIOCGIFCONF`, and then the subnet mask for each interface is obtained from a call to `ioctl()` using `SIOCGIFNETMASK`.

On FreeBSD the `ifap` argument to `getifaddrs()` is a pointer to a location to store a linked list of the interface information in, corresponding to the return type of the model getifaddrs().

15.9.4 Model details

Any of the errors possible when making an `ioctl()` call are possible: EIO, ENOTTY, ENXIO, and ENODEV. None of these are modelled.

Note that the Posix interface admits the possibility that the interfaces will change between the two calls, whereas in the model interface the getifaddrs() call is atomic.

15.9.5 Summary

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

15.9.6 Rules

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

$$h\ \ ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \xrightarrow{tid\cdot \mathsf{getifaddrs}()} h\ \ ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}\ iflist))_{\mathrm{sched\_timer}})$$

$$\begin{array}{l} ifidlist \in \mathrm{ORDERINGS}\ ifidset \wedge \\ \mathbf{length}\ ifidlist = \mathbf{length}\ iflist \wedge \\ \\ ifidset = \{(ifid, hifd) \mid \\ \qquad ifid \in \mathbf{dom}(h.ifds) \wedge \\ \qquad hifd = h.ifds[ifid]\} \wedge \\ \\ \mathbf{every}\ \mathbf{I}(\mathbf{map2}(\lambda(ifid, hifd)(ifid', primary, ipslist, netmask).(ifid' = ifid \wedge \\ \qquad\qquad primary = hifd.primary \wedge \\ \qquad\qquad ipslist \in \mathrm{ORDERINGS}\ hifd.ipset \wedge \\ \qquad\qquad netmask = hifd.netmask)) \\ \qquad ifidlist\ iflist) \end{array}$$

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 $tid\cdot$getifaddrs() transition is made, leaving the thread state $\textsc{Ret}(\mathrm{OK}\, iflist)$.

Variations

WinXP	This call does not exist on WinXP.

15.10 getpeername() (TCP and UDP)

$\mathsf{getpeername} : \mathsf{fd} \to (\mathsf{ip} * \mathsf{port})$

A call to getpeername(fd) returns the peer address of the socket referred to by file descriptor fd. If the file descriptor refers to a socket *sock* then a successful call will return (i_2, p_2) where $sock.is_2 = \uparrow i_2$, and $sock.ps_2 = \uparrow p_2$.

15.10.1 Errors

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

ENOTCONN	Socket not connected to a peer.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.10.2 Common cases

getpeername_1; *return_1*

15.10.3 API

Posix:	`int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`
FreeBSD:	`int getpeername(int s, struct sockaddr *name, socklen_t *namelen);`
Linux:	`int getpeername(int s, struct sockaddr *name, socklen_t *namelen);`
WinXP:	`int getpeername(SOCKET s,struct sockaddr* name, int* namelen);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to get the peer address of, corresponding to the fd argument in the model getpeername().
- `address` is a pointer to a `sockaddr` structure of length `address_len`, which contains the peer address of the socket upon return. These two correspond to the (ip $*$ port) return type of the model getpeername(). The `sin_addr.s_addr` field of the `address` structure holds the peer IP address, corresponding to the ip in the return tuple; the `sin_port` field of the `address` structure holds the peer port, corresponding to the port in the return tuple.
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

15.10.4 Model details

The following errors are not modelled:

- According to the FreeBSD man page for getpeername(), ECONNRESET can be returned if the connection has been reset by the peer. This behaviour has not been observed in any tests.
- On FreeBSD, Linux, and WinXP, EFAULT can be returned if the `name` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `getpeername()` that is excluded by the clean interface used in the model getpeername().
- In Posix, EINVAL can be returned if the socket has been shutdown; none of the implementations in the model return this error from a getpeername() call.
- In Posix, EOPNOTSUPP is returned if the getpeername() operation is not supported by the protocol. Both TCP and UDP support this operation.

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.10.5 Summary

getpeername_1	**all: fast succeed**	Successfully return socket's peer address
getpeername_2	**all: fast fail**	Fail with ENOTCONN: socket not connected to a peer

15.10.6 Rules

getpeername_1 <u>**all: fast succeed**</u> **Successfully return socket's peer address**

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

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

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 $tid\cdot$getpeername(fd) transition is made, leaving the thread state $\textsc{Ret}(\mathrm{OK}(i_2, p_2))$.

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 $p_2 = \textsc{Port}\ 0$. Additionally, if *sock* is a TCP socket then it may be in any state.

getpeername_2 **<u>all: fast fail</u>** **Fail with ENOTCONN: socket not connected to a peer**

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

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

$\qquad \mathrm{windows\_arch}\ \ h.arch))$

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.

15.11 getsockbopt() (TCP and UDP)

getsockbopt : (fd ∗ sockbflag) → bool

A call to getsockbopt(fd, *flag*) returns the value of one of the socket's boolean-valued flags.

The fd argument is a file descriptor referring to the socket to retrieve a flag's value from, and the *flag* argument is the boolean-valued socket flag to get. Possible flags are:

- SO_BSDCOMPAT Reports whether the BSD semantics for delivery of ICMPs to UDP sockets with no peer address set is enabled.
- SO_DONTROUTE Reports whether outgoing messages bypass the standard routing facilities.
- SO_KEEPALIVE Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.
- SO_OOBINLINE Reports whether the socket leaves received out-of-band data (data marked urgent) inline.

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

The return value of the getsockbopt() call is the boolean-value of the specified socket flag.

15.11.1 Errors

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

ENOPROTOOPT	The specified flag is not supported by the protocol.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.11.2 Common cases

getsockbopt_1; *return_1*

15.11.3 API

getsockbopt() is Posix `getsockopt()` for boolean-valued socket flags.

```
Posix:    int getsockopt(int socket, int level, int option_name,
           void *restrict option_value,
           socklen_t *restrict option_len);
FreeBSD:  int getsockopt(int s, int level, int optname,
           void *optval, socklen_t *optlen);
Linux:    int getsockopt(int s, int level, int optname,
           void *optval, socklen_t *optlen);
WinXP:    int getsockopt(SOCKET s,int level,int optname,
           char* optval, int* optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket on which to get the flag, corresponding to the fd argument of the model getsockbopt().
- `level` is the protocol level at which the flag resides: `SOL_SOCKET` for the socket level options, and `option_name` is the flag to be retrieved. These two correspond to the *flag* argument to the model getsockbopt() where the possible values of `option_name` are limited to: SO_BSDCOMPAT, SO_DONTROUTE, SO_KEEPALIVE, SO_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()`.

15.11.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to getsockbopt().
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.11.5 Summary

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

15.11.6 Rules

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

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

$$\begin{array}{l}
fd\ \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \textsc{File}(\mathrm{FT_}\textsc{Socket}(sid), ff) \wedge \\
sf = (h.socks[sid]).sf \wedge \\
(\mathrm{windows_arch}\ \ h.arch \wedge \mathrm{proto_of}(h.socks[sid]).pr = \mathrm{PROTO_UDP} \\
\quad \Longrightarrow\ f\ \notin \{\mathrm{SO_KEEPALIVE}; \mathrm{SO_OOBINLINE}\})
\end{array}$$

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 $\textsc{Ret}(\mathrm{OK}(sf.b(f)))$ 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**

$$\begin{array}{l}
\quad h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d); \\
\quad socks := socks \oplus \\
\qquad [(sid, sock\ \langle\!|pr := \mathrm{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle \\
\xrightarrow{tid\cdot\mathsf{getsockbopt}(fd,f)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{FAIL\ ENOPROTOOPT}))_{\mathrm{sched_timer}}); \\
\qquad socks := socks \oplus \\
\qquad\quad [(sid, sock\ \langle\!|pr := \mathrm{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle
\end{array}$$

$$\begin{array}{l}
\mathrm{windows_arch}\ \ h.arch \wedge \\
fd\ \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \textsc{File}(\mathrm{FT_}\textsc{Socket}(sid), ff) \wedge \\
f\ \in \{\mathrm{SO_KEEPALIVE}; \mathrm{SO_OOBINLINE}\}
\end{array}$$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the Run state, a getsockbopt(fd, f) call is made, where f is either SO_KEEPALIVE or SO_OOBINLINE. The call fails with an ENOPROTOOPT error.

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

Variations

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

15.12 getsockerr() (TCP and UDP)

getsockerr : fd → unit

A call getsockerr(fd) returns the pending error of a socket, clearing it, if there is one.

fd is a file descriptor referring to a socket. If the socket has a pending error then the getsockerr() call will fail with that error, otherwise it will return successfully.

15.12.1 Errors

In addition to failing with the pending error, a call to getsockerr() can fail with the errors below, in which case the corresponding exception is raised:

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

15.12.2 Common cases

getsockerr_1; *return_1*

getsockerr_2; *return_1*

15.12.3 API

getsockerr() is Posix `getsockopt()` for the `SO_ERROR` socket option.

Posix:
```
int getsockopt(int socket, int level, int option_name,
 void *restrict option_value,
 socklen_t *restrict option_len);
```
FreeBSD:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
Linux:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
WinXP:
```
int getsockopt(SOCKET s,int level,int optname,
 char* optval, int* optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to get the option on, corresponding to the fd argument of the model getsockerr().
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. For getsockerr() `option_name` is set to `SO_ERROR`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsockopt()`. When `option_name` is `SO_ERROR` these fields are not used.

- the returned `int` is either `0` to indicate the socket has no pending error or `-1` to indicate a pending error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

15.12.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, the flag for getsockerr() is always `SO_ERROR` so this error cannot occur.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.12.5 Summary

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

15.12.6 Rules

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

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\textsc{Run})_d)\rangle\!\rangle \xrightarrow{tid\cdot \mathsf{getsockerr}(fd)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

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

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*·getsockerr(*fd*) transition is made, leaving the thread state Ret(OK()).

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

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

$fd\ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\mathrm{FT\_SOCKET}(sid), ff) \wedge$
$\uparrow e = sock.es \wedge$
$sock' = sock\ \langle\!\langle\ es := *\rangle\!\rangle$

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.

15.13 getsocklistening() (TCP and UDP)

getsocklistening : fd → bool

A call to getsocklistening(fd) returns **T** if the socket referenced by fd is listening, or **F** otherwise. For TCP a socket is listening if it is in the LISTEN state. For UDP, which is not a connection-oriented protocol, a socket can never be listening.

15.13.1 Errors

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

ENOPROTOOPT	FreeBSD does not support this socket option, and on Linux and WinXP this option is not supported for UDP sockets.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.13.2 Common cases

getsocklistening_1; *return_1*

15.13.3 API

getsocklistening() is Posix `getsockopt()` for the `SO_ACCEPTCONN` socket option.

Posix:
```
int getsockopt(int socket, int level, int option_name,
 void *restrict option_value,
 socklen_t *restrict option_len);
```
FreeBSD:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
Linux:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
WinXP:
```
int getsockopt(SOCKET s,int level,int optname,
 char* optval, int* optlen);
```
In the Posix interface:

- `socket` is the file descriptor of the socket to get the option on, corresponding to the fd argument of the model getsocklistening().
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. For getsocklistening() `option_name` is set to `SO_ACCEPTCONN`.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `getsockopt()`. The value stored in the location corresponds to the bool return value of the model getsocklistening().
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The Linux and WinXP interfaces are similar except where noted. FreeBSD does not support the `SO_ACCEPTCONN` socket option.

15.13.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, the flag for getsocklistening() is always `SO_ACCEPTCONN` so this error cannot occur.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.13.5 Summary

getsocklistening_1	**tcp: fast succeed**	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

15.13.6 Rules

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

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize UN}})_d)\rangle\!\rangle \xrightarrow{tid\cdot \mathsf{getsocklistening}(fd)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize ET}}(\mathrm{OK}\ b))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

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

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 = **T** if the socket is in the LISTEN state, and b = **F** otherwise.

Variations

FreeBSD	This rule does not apply: see *getsocklistening_3*.

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

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize UN}})_d)\rangle\!\rangle \xrightarrow{tid\cdot \mathsf{getsocklistening}(fd)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{R{\scriptsize ET}}(\mathrm{FAIL\ ENOPROTOOPT}))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

$\mathrm{bsd\_arch}\ \ h.arch \wedge$

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

Description

On FreeBSD, a getsocklistening(fd) call is made from thread tid which is in the RUN state wherefd 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**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid\cdot \text{getsocklistening}(fd)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(ret))_{\text{sched\_timer}})\rangle\!\rangle$$

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

Description

Consider a UDP socket sid, referenced by fd. From thread tid, which is in the RUN state, a getsocklistening(fd) call is made. On Linux the call succeeds, returning **F**; on FreeBSD and WinXP the call fails with an ENOPROTOOPT error.

A tid·getsocklistening(fd) transition is made, leaving the thread state RET(OK(**F**)) on Linux, and RET(FAIL ENOPROTOOPT) on FreeBSD and Linux.

Variations

Posix	As above: the call fails with an ENOPROTOOPT error.
FreeBSD	As above: the call fails with an ENOPROTOOPT error.
Linux	As above: the call succeeds, returning **F**.
WinXP	As above: the call fails with an ENOPROTOOPT error.

15.14 getsockname() (TCP and UDP)

getsockname : fd → (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 $(sock.is_1, sock.ps_1)$.

15.14.1 Errors

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

ECONNRESET	On FreeBSD, TCP socket has its *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.

15.14.2 Common cases

getsockname_1; *return_1*

15.14.3 API

Posix:	`int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`
FreeBSD:	`int getsockname(int s, struct sockaddr *name, socklen_t *namelen);`
Linux:	`int getsockname(int s, struct sockaddr *name, socklen_t *namelen);`
WinXP:	`int getsockname(SOCKET s, struct sockaddr* name, int* namelen);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to get the local address of, corresponding to the fd argument in the model getsockname().
- `address` is a pointer to a `sockaddr` structure of length `address_len`, which contains the local address of the socket upon return. These two correspond to the (ip option, port option) return type of the model getsockname(). If the `sin_addr.s_addr` field of the `name` structure is set to `0` on return, then the socket's local IP address is not set: the ip option member of the return tuple is set to $*$; otherwise, if it is set to `i` then it corresponds to the socket having local IP address and so the ip option member of the return tuple is$\uparrow i$. If the `sin_port` field of the `name` structure is set to `0` on return then the socket does not have a local port set, corresponding to the port option in the return tuple being $*$; otherwise the `sin_port` field is set to `p` corresponding to the socket having its local port set: the port option in the return tuple is $\uparrow p$.
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

15.14.4 Model details

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, EFAULT can be returned if the `name` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `getsockname()` that is excluded by the clean interface used in the model getsockname().
- in Posix, EINVAL can be returned if the socket has been shutdown. None of the implementations return EINVAL in this case.

- in Posix, EOPNOTSUPP is returned if the getsockname() operation is not supported by the protocol. Both UDP and TCP support this operation.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.14.5 Summary

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

15.14.6 Rules

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

$$h \; \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d)|\!\rangle \xrightarrow{tid \cdot \mathsf{getsockname}(fd)} h \; \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{OK}(sock.is_1, sock.ps_1)))_{\mathrm{sched\_timer}})|\!\rangle$$

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

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 $\mathrm{RET}(\mathrm{OK}(sock.is_1, sock.ps_1))$.

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.
WinXP	As above with the additional condition that either the socket's local IP address or local port must be set.

getsockname_2 **tcp: fast fail** **Fail with ECONNRESET: previous connection attempt has failed on FreeBSD**

$h \; \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);$
$socks := socks \oplus [(sid, sock)]|\!\rangle$

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

$\text{bsd\_arch}\ \ h.arch \wedge$
$sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$(tcp\_sock.cb.bsd\_cantconnect = \mathbf{T} \wedge sock.ps_1 = *) \wedge$

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

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**

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

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

$\text{windows\_arch}\ \ h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff)$

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.

15.15 getsocknopt() (TCP and UDP)

getsocknopt : (fd ∗ socknflag) → int

A call to getsocknopt(fd, *flag*) returns the value of one of the socket's numeric flags. The fd argument is a file descriptor referring to the socket to retrieve a flag's value from. The *flag* argument is a numeric socket flag. Possible flags are:

- SO_RCVBUF Reports receive buffer size information.
- SO_RCVLOWAT Reports the minimum number of bytes to process for socket input operations.
- SO_SNDBUF Reports send buffer size information.
- SO_SNDLOWAT Reports the minimum number of bytes to process for socket output operations.

The return value of the getsocknopt() call is the numeric-value of the specified *flag*.

15.15.1 Errors

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

ENOPROTOOPT	The specified flag is not supported by the protocol.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.15.2 Common cases

getsocknopt_1; *return_1*

15.15.3 API

getsocknopt() is Posix `getsockopt()` for numeric socket flags.

Posix:
```
int getsockopt(int socket, int level, int option_name,
 void *restrict option_value,
 socklen_t *restrict option_len);
```
FreeBSD:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
Linux:
```
int getsockopt(int s, int level, int optname,
 void *optval, socklen_t *optlen);
```
WinXP:
```
int getsockopt(SOCKET s,int level,int optname,
 char* optval, int* optlen);
```

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the fd argument of the model getsocknopt().
- `level` is the protocol level at which the option resides: `SOL_SOCKET` for the socket level options, and `option_name` is the option to be retrieved. These two correspond to the *flag* argument to the model getsocknopt() where the possible values of `option_name` are limited to SO_RCVBUF, SO_RCVLOWAT, SO_SNDBUF and SO_SNDLOWAT.
- `option_value` is a pointer to a location of size `option_len` to store the value retrieved by `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()`.

15.15.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to getsocknopt().
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.15.5 Summary

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

15.15.6 Rules

getsocknopt_1 **all: fast succeed** **Successfully retrieve value of a numeric socket flag**

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

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\mathrm{FT\_}\textsc{Socket}(sid), ff) \wedge$
$sf = (h.socks[sid]).sf \wedge$
$(\mathrm{windows\_arch}\ h.arch \implies f \notin \{\mathrm{SO\_RCVLOWAT}; \mathrm{SO\_SNDLOWAT}\})$

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 $\textsc{Ret}(\mathrm{OK}(\mathbf{int\_of\_num}(sf.n(f))))$.

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**

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

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

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*·getsocknopt(fd, f) transition is made, leaving the thread state RET(FAIL ENOPROTOOPT).

Variations

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

15.16 getsocktopt() (TCP and UDP)

getsocktopt : (fd $*$ socktflag) $\rightarrow$ (int $*$ int) 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.

15.16.1 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.

15.16.2 Common cases

getsocktopt_1; *return_1*

15.16.3 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()`.

15.16.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to 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.

15.16.5 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

15.16.6 Rules

getsocktopt_1 **<u>all: fast succeed</u>** **Successfully retrieve value of time-option socket flag**

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

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

$\neg(\text{windows\_arch } h.arch \wedge \text{proto\_of}(h.socks[sid]).pr = \text{PROTO\_UDP} \wedge$
$f = \text{SO\_LINGER})$

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 $\text{OK}(t)$ where t is the value of the socket's flag f.

A $tid\cdot$getsocktopt(fd, f) transition is made, leaving the thread state $\text{Ret}(\text{OK}t)$.

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**

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

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

Description

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

A $tid\cdot$getsocktopt(fd, f) transition is made, leaving the thread state $\text{Ret}(\text{ENOPROTOOPT})$.

Variations

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

15.17 listen() (TCP only)

listen : fd ∗ int → unit

A call to listen(fd, n) puts a TCP socket that is in the CLOSED state into the LISTEN state, making it a passive socket, so that incoming connections for the socket will be accepted by the host and placed on its listen queue. Here fd is a file descriptor referring to the socket to put into the LISTEN state and n is the *backlog* used to calculate the maximum lengths of the two components of the socket's listen queue: its pending connections queue, $lis.q_0$, 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.

15.17.1 Errors

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

EADDRINUSE	Another socket is listening on this local port.
EINVAL	On FreeBSD the socket has been shutdown for writing; on Linux the socket is not in the CLOSED or LISTEN state; or on WinXP the socket is not bound,
EISCONN	On WinXP the socket is already connected: it is not in the CLOSED or LISTEN state.
EOPNOTSUPP	The 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.

15.17.2 Common cases

A TCP socket is created, has its local address and port set by bind(), and then is put into the LISTEN state which can accept new incoming connections: *socket_1*; *return_1*; *bind_1 return_1*; *listen_1*; *return_1*; ...

15.17.3 API

Posix:	`int listen(int socket, int backlog);`
FreeBSD:	`int listen(int s, int backlog);`
Linux:	`int listen(int s, int backlog);`
WinXP:	`int listen(SOCKET s, int backlog);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to put into the LISTEN state, corresponding to the fd argument of the model listen().
- `backlog` is an `int` on which the maximum permitted length of the socket's listen queue depends. It corresponds to the n argument of the model listen().
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

15.17.4 Model details

The following errors are not modelled:

- In Posix, EACCES may be returned if the calling process does not have the appropriate privileges. This is not modelled here.
- In Posix, EDESTADDRREQ shall be returned if the socket is not bound to a local address and the protocol does not support listening on an unbound socket. WinXP returns an EINVAL error in this case; FreeBSD and Linux autobind the socket if listen() is called on an unbound socket.

- WSAEINPROGRESS is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.17.5 Summary

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

15.17.6 Rules

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

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(\text{CLOSED}, cb, *, [\,], *, [\,], *, \text{NO\_OOBDATA})))];\\
\quad \mathsf{listen} := listen_0|\!\rangle\\
\xrightarrow{tid\cdot\mathsf{listen}(fd,\,n)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK}()))_{\text{sched\_timer}});\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \text{NO\_OOBDATA})))];\\
\quad \mathsf{listen} := sid :: listen_0;\\
\quad bound := bound|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge\\
fid = h.fds[fd] \wedge\\
h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge\\
(\text{bsd\_arch}\ h.arch \vee cantrcvmore = \mathbf{F}) \wedge\\
\neg(\text{windows\_arch}\ h.arch \wedge IS\_NONE\ ps_1) \wedge\\
(\text{bsd\_arch}\ h.arch \implies cb.bsd\_cantconnect = \mathbf{F}) \wedge\\
p_1 \in \text{autobind}(ps_1, \text{PROTO\_TCP}, socks \backslash\backslash sid) \wedge\\
(\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h.bound\ \mathbf{else}\ bound = h.bound) \wedge\\
lis = \langle\!|\ q_0 := [\,];\\
\qquad q := [\,];\\
\qquad qlimit := n|\!\rangle
\end{array}
$$

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.

Rule version: $Id: TCP1_hostLTSScript.sml,v 1.961 2005/03/18 10:34:36 kw217 Exp $

The call succeeds: a $tid\cdot\mathsf{listen}(fd, n)$ transition is made, leaving the thread in state $\text{RET}(\text{OK}())$. The socket is put in the LISTEN state, with an empty listen queue, *lis*, with n as its backlog. *sid* is added to the host's list of listening sockets, $\mathsf{listen} := sid :: listen_0$, 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* (p195).

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

$$\begin{array}{l} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\\ \quad socks := socks \oplus\\ \qquad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\ \qquad\qquad \text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \text{NO\_OOBDATA})))];\\ \quad \mathsf{listen} := listen_0|\!\rangle \end{array}$$

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

$$\begin{array}{l} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}});\\ \quad socks := socks \oplus\\ \qquad [(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{F}, cantrcvmore,\\ \qquad\qquad \text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis', [\,], *, [\,], *, \text{NO\_OOBDATA})))];\\ \quad \mathsf{listen} := sid :: listen_0|\!\rangle \end{array}$$

$$\begin{array}{l} fd \in \mathbf{dom}(h.fds) \wedge\\ fid = h.fds[fd] \wedge\\ h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge\\ (\text{bsd\_arch}\ h.arch \vee cantrcvmore = \mathbf{F}) \wedge\\ lis' = lis\ \langle\!|\ qlimit := n|\!\rangle \end{array}$$

Description

From thread *tid*, which is in the RUN state, a $\mathsf{listen}(fd, n)$ 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 $tid\cdot\mathsf{listen}(fd, n)$ transition is made, leaving the thread state $\text{RET}(\text{OK}())$. The backlog value of the socket's listen queue, *lis.qlimit* is updated to be n, resulting in a new listen queue lis' for the socket. *sid* is added to the head of the host's listen queue, $\mathsf{listen} := sid :: listen_0$.

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

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

$$\begin{array}{l} \text{bsd\_arch}\ h.arch \wedge\\ fd \in \mathbf{dom}(h.fds) \wedge\\ fid = h.fds[fd] \wedge \end{array}$$

$h.\mathit{files}[\mathit{fid}] = \text{FILE}(\text{FT\_SOCKET}(\mathit{sid}), \mathit{ff}) \wedge$
$\mathit{sock} = \text{SOCK}(\uparrow \mathit{fid}, \mathit{sf}, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, \mathit{es}, \mathit{cantsndmore}, \mathit{cantrcvmore}, \text{TCP\_PROTO}(\mathit{tcp\_sock})) \wedge$
$\mathit{tcp\_sock}.\mathit{st} \notin \{\text{CLOSED}; \text{LISTEN}\} \wedge$
$\mathit{sock}' = \mathit{sock} \ \langle\!|\ \mathit{pr} := \text{TCP\_PROTO}(\mathit{tcp\_sock} \ \langle\!|\ \mathit{st} := \text{LISTEN}; \mathit{lis} := \uparrow \mathit{lis}|\!\rangle)|\!\rangle \wedge$
$\mathit{lis} = \langle\!|\ q_0 := [\,];$
$\quad q := [\,];$
$\quad \mathit{qlimit} := n|\!\rangle$

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*·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**

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

$\text{windows\_arch } h.\mathit{arch} \wedge$
$\mathit{fd} \in \mathbf{dom}(h.\mathit{fds}) \wedge$
$\mathit{fid} = h.\mathit{fds}[\mathit{fd}] \wedge$
$h.\mathit{files}[\mathit{fid}] = \text{FILE}(\text{FT\_SOCKET}(\mathit{sid}), \mathit{ff}) \wedge$
$h.\mathit{socks}[\mathit{sid}] = \mathit{sock} \wedge$
$\text{proto\_of } \mathit{sock}.\mathit{pr} = \text{PROTO\_TCP} \wedge$
$\mathit{sock}.\mathit{ps}_1 = *$

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: $\mathit{sock}.\mathit{ps}_1 = *$. 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**

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

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

$\neg(\text{bsd\_arch } h.arch) \wedge$
$(\textbf{if } \text{windows\_arch } h.arch \textbf{ then}$
$\quad err = \text{EISCONN}$
$\textbf{else if } \text{linux\_arch } h.arch \textbf{ then}$
$\quad err = \text{EINVAL}$
$\textbf{else}$
$\quad \textbf{F})$

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**

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

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

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**

$$\begin{array}{l} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\\ \quad socks := socks \oplus\\ \qquad [(sid, sock\ \langle\!|cantsndmore := cantsndmore; pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!|st := st|\!\rangle)|\!\rangle)]|\!\rangle\\ \xrightarrow{tid\cdot\mathsf{listen}(fd,n)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched\_timer}});\\ \quad socks := socks \oplus\\ \qquad [(sid, sock\ \langle\!|cantsndmore := cantsndmore; pr := \text{TCP\_PROTO}(tcp\_sock\ \langle\!|st := st|\!\rangle)|\!\rangle)]|\!\rangle \end{array}$$

$$\begin{array}{l} \text{bsd\_arch}\ h.arch \wedge\\ fd \in \mathbf{dom}(h.fds) \wedge\\ fid = h.fds[fd] \wedge\\ st \in \{\text{CLOSED}; \text{LISTEN}\} \wedge\\ h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge\\ (cantsndmore = \mathbf{T} \vee tcp\_sock.cb.bsd\_cantconnect = \mathbf{T}) \end{array}$$

Description

On FreeBSD, from thread *tid*, which is in the RUN state, a $\mathsf{listen}(fd, n)$ 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 $tid\cdot\mathsf{listen}(fd, n)$ transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

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

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

$$\begin{array}{l} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle\\ \xrightarrow{tid\cdot\mathsf{listen}(fd,n)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EOPNOTSUPP}))_{\text{sched\_timer}})|\!\rangle \end{array}$$

$$\begin{array}{l} fd \in \mathbf{dom}(h.fds) \wedge\\ fid = h.fds[fd] \wedge\\ h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge\\ \text{proto\_of}(h.socks[sid]).pr = \text{PROTO\_UDP} \end{array}$$

Description

Consider a UDP socket *sid*, referenced by *fd*. From thread *tid*, which is in the RUN state, a $\mathsf{listen}(fd, n)$ call is made. The call fails with an EOPNOTSUPP error.

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

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

15.18 pselect() (TCP and UDP)

$\mathsf{pselect} : (\mathsf{fd\ list} * \mathsf{fd\ list} * \mathsf{fd\ list} * (\mathsf{int} * \mathsf{int})\ \mathsf{option} * \mathsf{signal\ list\ \ option}) \rightarrow (\mathsf{fd\ list} * (\mathsf{fd\ list} * \mathsf{fd\ list}))$

A call to $\mathsf{pselect}(\mathit{readfds}, \mathit{writefds}, \mathit{exceptfds}, \mathit{timeout}, \mathit{sigmask})$ waits for one of the file descriptors in *readfds* to be ready for reading, *writefds* to be ready for writing, *exceptfds* to have a pending error, or for *timeout* to expire.

The *readfds* argument is a set of file descriptors to be checked for being ready to read. Broadly, a file descriptor fd is ready for reading if a $\mathsf{recv}(\mathsf{fd}, \_, \_)$ call on the socket would not block, i.e. if there is data present or a pending error.

The *writefds* argument is a set of file descriptors to be checked for being ready to write. Broadly, a file descriptor fd is ready for writing if a $\mathsf{send}(\mathsf{fd}, \_, \_, \_)$ call would not block.

The *exceptfds* argument is a set of file descriptors to be checked for exception conditions pending. A file descriptor fd has an exception condition pending if there exists out-of-band data for the socket it refers to or the socket is still at the out-of-band mark.

The *timeout* argument specifies how long the pselect() call should block waiting for a file descriptor to be ready. If $\mathit{timeout} = *$ then the call should block until one of the file descriptors in the *readfds*, *writefds*, or *exceptfds* becomes ready. If $\mathit{timeout} = \uparrow(s, ns)$ then the call should block for at most s seconds and ns nanoseconds. However, system activity can lengthen the timeout interval by an indeterminate amount.

The *sigmask* argument is used to set the signal mask, the set of signals to be blocked. In the implementations, if $\mathit{sigmask} = \uparrow(\mathit{siglist})$ then pselect() first replaces the current signal mask by *siglist* before proceeding with the call, and then restores the original signal mask upon return. This specification does not model the dynamic behaviour of signals, however, and so we specify the behaviour of pselect() only for an empty signal mask.

A return value of $(\mathit{readfds}', (\mathit{writefds}', \mathit{exceptfds}'))$ from a pselect() call signifies that: the file descriptors in $\mathit{readfds}'$ are ready for reading; the file descriptors in $\mathit{writefds}'$ are reading for writing; and the file descriptors in $\mathit{exceptfds}'$ have exceptional conditions pending.

If a $\mathsf{pselect}([\,], [\,], [\,], \mathit{Some}(s, ns), \mathit{sigmask})$ call is made then the call will block for s seconds and ns nanoseconds or until a signal occurs.

To perform a poll, a $\mathsf{pselect}(\mathit{readfds}, \mathit{writefds}, \mathit{exceptfds}, \mathit{Some}(0, 0), \mathit{sigmask})$ call should be made.

15.18.1 Errors

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

EBADF	One or more of the file descriptors in a set is not a valid file descriptor.
EINVAL	Time-out not well-formed, file descriptor out of range, or on WinXP all file descriptor sets are empty.
ENOTSOCK	One or more of the file descriptors in a set is not a valid socket.
EINTR	The system was interrupted by a caught signal.

15.18.2 Common cases

pselect() is called and returns immediately: *pselect_1*; *return_1*

pselect() blocks and then times out before any of the file descriptors become ready: *pselect_2*; *pselect_3*; *return_1* pselect() blocks, TCP data is received from the network and processed, making a file descriptor ready for reading, and then pselect() returns: *pselect_1*; *deliver_in_99*; *deliver_in_3*; *pselect_2*; *return_1*

pselect() blocks, UDP data is received from the network and processed, making a file descriptor ready for reading, and then pselect() returns: *pselect_1*; *deliver_in_99*; *deliver_in_udp_1*; *pselect_2*; *return_1*

pselect() blocks, TCP data is sent to the network, an acknowledgement is received and processed, making a file descriptor ready for writing, and then pselect() returns: *pselect_1*; *deliver_out_1*; *deliver_out_99*; *deliver_in_99*; *deliver_in_3*; *pselect_2*; *return_1*

15.18.3 API

Posix:
```
int pselect(int nfds, fd_set *restrict readfds,
 fd_set *restrict writefds, fd_set *restrict errorfds,
 const struct timespec *restrict timeout,
 const sigset_t *restrict sigmask);
```
FreeBSD:
```
int select(int nfds, fd_set *readfds, fd_set *writefds,
 fd_set *exceptfds, struct timeval *timeout);
```
Linux:
```
int pselect(int n, fd_set *readfds, fd_set *writefds,
 fd_set *exceptfds, const struct timespec *timeout,
 const sigset_t *sigmask);
```
WinXP:
```
int select(int nfds, fd_set* readfds, fd_set* writefds,
 fd_set* exceptfds, const struct timeval* timeout);
```

In the Posix interface:

- `nfds` specifies the range of file descriptors to be tested. The first `nfds` file descriptors shall be checked in each set. This is not necessary in the model pselect() as the file descriptor sets are implemented as a list rather than the integer arrays in Posix `pselect()`.
- `readfds` on input specifies the file descriptors to be checked for being ready to read, corresponding to the *readfds* argument of the model pselect(). On output `readfds` indicates which of the file descriptors specified on input are ready to read, corresponding to the first fd list in the return type of the model pselect(). An `fd_set` is an integer array, where each bit of each integer corresponds to a file descriptor. If that bit is set then that file descriptor should be checked. `FD_CLR()`, `FD_ISSET()`, `FD_SET()`, and `FD_ZERO()` are provided to set bits in an `fd_set`.
- `writefds` on input specifies the file descriptors to be checked for being ready to write, corresponding to the *writefds* argument of the model pselect(). On output `writefds` indicates which of the file descriptors specified on input are ready to write, corresponding to the second fd list in the return type of the model pselect().
- `errorfds` on input specifies the file descriptors to be checked for pending error conditions, corresponding to the *exceptfds* argument of the model pselect(). On output `exceptfds` indicated which of the file descriptors specified on input have pending error conditions, corresponding to the third fd list in the return type of the model pselect().
- `timeout` specifies how long the `pselect()` call shall block before timing out, corresponding to the *timeout* argument of the model pselect(). If the `timeout` parameter is a null pointer this corresponds to $timeout = *$; if the `timeout` parameter is not a null pointer, then its two fields, `timeout.tv_sec` (the number of seconds) and `timeout.tv_nsec` (the number of nano-seconds), correspond to $timeout = \uparrow(s, ns)$ where s is the number of seconds, and ns is the number of nano-seconds.
- `sigmask` is the signal-mask to be used when examining the file descriptors, corresponding to the *sigmask* argument of the model pselect(). If `sigmask` is a null pointer then $sigmask = *$ in the model; if `sigmask` is not a null pointer then $sigmask = \uparrow sigs$ in the model where $sigs$ is the signal-mask to use.
- if the call is successful then the returned `int` is the number of bits set in the three `fd_set` arguments: the total number of file descriptors ready for reading, writing, or having exceptional conditions pending. Otherwise, the returned `int` is `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The Linux interface is similar. On FreeBSD and WinXP there is no `pselect()` call, only a `select()` call which is the same as the interface described above, except without the `sigmask` argument. The `select()` call corresponds to calling the model pselect() with $sigmask = *$. Additionally, the `timeout` argument is a pointer to a `timeval` structure which has two members `tv_sec` and `tv_usec`, specifying the seconds and micro-seconds to block for, rather than seconds and nano-seconds.

The FreeBSD man page for `select()` warns of the following bug: "Version 2 of the Single UNIX Specification ("SUSv2") allows systems to modify the original timeout in place. Thus, it is unwise to assume that the timeout value will be unmodified by the select() call."

15.18.4 Model details

If the pselect() call blocks then the thread enters state PSELECT2(*readfds*, *writefds*, *exceptfds*) where:

- *readfds* : fd list is the list of file descriptors to be checked for being ready to read.
- *writefds* : fd list is the list of file descriptors to be checked for being ready to write.
- *exceptfds* : fd list is the list of file descriptors to be checked for pending exceptional conditions.

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.18.5 Summary

pselect_1	**all: fast succeed**	One or more file descriptors immediately ready, or no timeout set
soreadable		check whether a socket is readable
sowriteable		check whether a socket is writable
soexceptional		check whether a socket is exceptional
pselect_2	**all: block**	Normal case
pselect_3	**all: slow nonurgent succeed**	Something becomes ready or pselect times out
pselect_4	**all: fast fail**	Fail with EINVAL: Timeout not well-formed
pselect_5	**all: fast fail**	Fail with EINVAL: File descriptor out of range
pselect_6	**all: fast fail**	Fail with EBADF or ENOTSOCK: Bad file descriptor

15.18.6 Rules

pselect_1 **all: fast succeed** **One or more file descriptors immediately ready, or no timeout set**

$$h \,\langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d) \rangle\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}$$

$$h \,\langle\!\langle ts := ts \oplus (tid \mapsto \left(\mathrm{RET}(\mathrm{OK}(readfds'', writefds'', exceptfds''))\right)_{\mathrm{sched\_timer}}) \rangle\!\rangle$$

$(\mathrm{tltimeopt\_wf}\ \ timeout \lor \mathrm{windows\_arch}\ \ h.arch) \land$
$sigmask = * \land$
$\neg(\exists fd\ n.(fd\ \in\ readfds \lor fd\ \in\ writefds \lor fd\ \in\ exceptfds) \land$
$\quad \textbf{if}\ \ \mathrm{windows\_arch}\ \ h.arch$
$\quad\ \textbf{then}\ \ n = (\textbf{max}(\textbf{length}\ \ readfds)(\textbf{max}(\textbf{length}\ \ writefds)(\textbf{length}\ \ exceptfds))) \land$
$\qquad\quad n \geq (\mathrm{FD\_SETSIZE}\ \ h.arch)$
$\quad\ \textbf{else}$
$\quad fd = \mathrm{FD}\ n \land$
$\quad n \geq \mathrm{FD\_SETSIZE}\ \ h.arch) \land$
$badreadfds = \textbf{filter}(\lambda fd.fd\ \ \notin \textbf{dom}(h.fds))readfds \land$
$badwritefds = \textbf{filter}(\lambda fd.fd\ \ \notin \textbf{dom}(h.fds))writefds \land$

$badexceptfds = \mathbf{filter}(\lambda fd.fd \notin \mathbf{dom}(h.fds))exceptfds\ \wedge$
$(\text{bsd\_arch}\ \ h.arch\ \vee$
$(badreadfds = [\,] \wedge badwritefds = [\,] \wedge badexceptfds = [\,])) \wedge$
$\neg(\exists fd.(fd\ \in\ readfds \vee fd\ \in\ writefds \vee fd\ \in\ exceptfds) \wedge$
$\quad\quad fd\ \notin \mathbf{dom}(h.fds)) \wedge$
$readfds' = \mathbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd\ \in \mathbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{soreadable}\ \ h.arch\ sock)readfds\ \wedge$
$writefds' = \mathbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd\ \in \mathbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{sowriteable}\ \ h.arch\ sock)writefds\ \wedge$
$exceptfds' = \mathbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd\ \in \mathbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{soexceptional}\ \ h.arch\ sock)exceptfds\ \wedge$
$(\text{windows\_arch}\ \ h.arch\ \implies\ readfds \neq [\,] \wedge writefds \neq [\,] \wedge exceptfds \neq [\,]) \wedge$
$(readfds' \neq [\,] \vee writefds' \neq [\,] \vee exceptfds' \neq [\,] \vee timeout = \uparrow(0,0)) \wedge$
$\mathbf{if}\ \ \text{windows\_arch}\ \ h.arch\ \ \mathbf{then}$
$\quad\quad readfds'' = readfds' \wedge writefds'' = writefds' \wedge exceptfds'' = exceptfds'$
$\mathbf{else}$
$\quad\quad readfds'' = \text{INSERT\_ORDERED}\ \ readfds'\ readfds\ badreadfds\ \wedge$
$\quad\quad writefds'' = \text{INSERT\_ORDERED}\ \ writefds'\ writefds\ badwritefds\ \wedge$
$\quad\quad exceptfds'' = \text{INSERT\_ORDERED}\ \ exceptfds'\ exceptfds\ badexceptfds$

Description

From thread *tid*, which is in the RUN state, a pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) call is made. The time-out is well-formed and no signal mask was set: $sigmask = *$. All of the file descriptors in the sets *readfds*, *writefds*, and *exceptfds* are greater than the maximum allowed file descriptor in a set for the architecure, FD_SETSIZE, and all of them are valid file descriptors: they are in the host's finite map of file descriptors, *h.fds*.

The call returns, without blocking, three sets: $readfds''$, $writefds''$, and $exceptfds''$. $readfds''$ is the set of valid file descriptors in *readfds* that are ready for reading: a blocking recv(fd, _, _) call would not block; see soreadable (p202) for details. $writefds''$ is the set of valid file descriptors in *writefds* that are ready for writing: a blocking send(fd, _, _) call would not block; see sowriteable (p202) for details. $exceptfds''$ is the set of valid file descriptors in *exceptfds* that have pending exceptional conditions; see soexceptional (p203) for details.

One of these three sets must be non-empty or else a zero timeout was specified, $timeout = \uparrow(0,0)$. A $tid \cdot$pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) transition is made, leaving the thread state RET(OK($readfds''$, $writefds''$, $exceptfds''$)).

Variations

FreeBSD	Invalid file descriptors (ones not in the host's finite map of file descriptors, *h.fds*) may be present in the sets *readfds*, *writefds*, and *exceptfds*, and all such file descriptors will then be included in the return sets $readfds''$, $writefds''$, and $exceptfds''$.
WinXP	On WinXP FD_SETSIZE is the maximum number of file descriptors in a set, so none of the sets *readfds*, *writefds*, and *exceptfds* has more than FD_SETSIZE members. Additionally, all three sets may not be empty. The time-out need not be well-formed because one or more file descriptors is immediately ready.

– **check whether a socket is readable :**

soreadable $arch$ $sock$ =
case $sock.pr$ **of**
TCP_PROTO(tcp) →
  (**length** $tcp.rcvq \geq sock.sf.n(\mathrm{SO\_RCVLOWAT}) \vee$
  $sock.cantrcvmore \vee$
  (linux_arch $arch \wedge tcp.st = \mathrm{CLOSED}) \vee$
  $(tcp.st = \mathrm{LISTEN} \wedge$
    $\exists lis.tcp.lis = \uparrow lis \wedge$
      $lis.q \neq [\,]) \vee$
  $sock.es \neq *)$ ‖
UDP_PROTO(udp) →
  $(udp.rcvq \neq [\,] \vee sock.es \neq * \vee (sock.cantrcvmore \wedge \neg$ 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.

– **check whether a socket is writable :**

sowriteable $arch$ $sock$ =
case $sock.pr$ **of**
TCP_PROTO(tcp) →
  $((tcp.st \in \{\mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}\} \wedge$
    $sock.sf.n(\mathrm{SO\_SNDBUF}) -$ **length** $tcp.sndq \geq sock.sf.n(\mathrm{SO\_SNDLOWAT})) \vee$ (* change to send_buffer_space *)
  (**if** linux_arch $arch$ **then** $\neg sock.cantsndmore$ **else** $sock.cantsndmore) \vee$
  (linux_arch $arch \wedge tcp.st = \mathrm{CLOSED}) \vee$
  $sock.es \neq *)$ ‖
UDP_PROTO(udp) → **T**

Variations

Linux	On all OSes, attempting to write to a closed socket yields an immediate error. Only on Linux, however, does 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).

– **check whether a socket is exceptional :**
soexceptional *arch sock* =
case *sock.pr* **of**
TCP_PROTO(tcp) $\rightarrow$
  ($tcp.st$ = ESTABLISHED $\wedge$
  ($tcp.rcvurp = \uparrow 0 \vee$
    ($\exists c. tcp.iobc$ = OOBDATA c))) $\|$
UDP_PROTO(udp) $\rightarrow$ **F**

Description

A TCP socket has a pending exceptional condition if it is in state ESTABLISHED and has a pending byte of out-of-band data.

A UDP socket never has a pending exceptional condition.

pselect_2 **all: block** **Normal case**

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Run})_d) |\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{pselect}(readfds, writefds, exceptfds, timeout, sigmask)}$$

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\textsc{PSelect2}(readfds, writefds, exceptfds))_{\text{kern\_timer } d'}) |\!\rangle$$

tltimeopt_wf *timeout* $\wedge$
$d' = \mathbf{min}(\text{time\_of\_tltimeopt } timeout)\, \text{pselect\_timeo\_t\_max} \wedge$
$sigmask = * \wedge$
$\neg(\exists fd\ n.(fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$
  if windows_arch $h.arch$
  then $n = \mathbf{max}(\mathbf{length}\ readfds)(\mathbf{max}(\mathbf{length}\ writefds)(\mathbf{length}\ exceptfds)) \wedge$
    $n \geq$ FD_SETSIZE $h.arch$
  else
  fd = FD $n \wedge$
  $n \geq$ FD_SETSIZE $h.arch) \wedge$
$\neg(\exists fd.(fd \in readfds \vee fd \in writefds \vee fd \in exceptfds) \wedge$
  $fd \notin \mathbf{dom}(h.fds)) \wedge$
(windows_arch $h.arch \implies readfds \neq [\,] \wedge writefds \neq [\,] \wedge exceptfds \neq [\,])$

Description

From thread *tid*, which is in the Run state, a pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) call is made. The time-out is well-formed and no signal mask was set: $sigmask = *$. All of the file descriptors in the sets *readfds*, *writefds*, and *exceptfds* are greater than the maximum allowed file descriptor in a set for the architecure, FD_SETSIZE, and all of them are valid file descriptors: they are in the host's finite map of file descriptors, *h.fds*.

The call blocks: a *tid*·pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) transition is made, leaving the thread state PSelect2(*readfds*, *writefds*, *exceptfds*).

Variations

WinXP	On WinXP FD_SETSIZE is the maximum number of file descriptors in a set, so none of the sets *readfds*, *writefds*, and *exceptfds* has more than FD_SETSIZE members. Additionally, all three sets may not be empty.

pselect_3 **all: slow nonurgent succeed** **Something becomes ready or pselect times out**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{PSELECT2}(readfds, writefds, exceptfds))_d)|\!\rangle$$
$$\xrightarrow{\tau}\ h\ \langle\!|ts := ts \oplus (tid \mapsto \big(\text{RET}(\text{OK}(readfds'', writefds'', exceptfds''))\big)_{\text{sched\_timer}})|\!\rangle$$

$readfds' = \textbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd \in \textbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{soreadable}\ \ h.arch\ sock)readfds \wedge$
$writefds' = \textbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd \in \textbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{sowriteable}\ \ h.arch\ sock)writefds \wedge$
$exceptfds' = \textbf{filter}(\lambda fd.\exists fid\ ff\ sid\ sock.$
$\quad fd \in \textbf{dom}(h.fds) \wedge$
$\quad fid = h.fds[fd] \wedge$
$\quad h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\quad sock = h.socks[sid] \wedge$
$\quad \text{soexceptional}\ \ h.arch\ sock)exceptfds \wedge$
$(readfds' \neq [\,] \vee writefds' \neq [\,] \vee exceptfds' \neq [\,] \vee \text{timer\_expires}\ \ d) \wedge$
$badreadfds = \textbf{filter}(\lambda fd.fd\ \notin \textbf{dom}(h.fds))readfds \wedge$
$badwritefds = \textbf{filter}(\lambda fd.fd\ \notin \textbf{dom}(h.fds))writefds \wedge$
$badexceptfds = \textbf{filter}(\lambda fd.fd\ \notin \textbf{dom}(h.fds))exceptfds \wedge$
$\textbf{if}\ \ \text{windows\_arch}\ \ h.arch\ \ \textbf{then}$
$\quad readfds'' = readfds' \wedge writefds'' = writefds' \wedge exceptfds'' = exceptfds'$
$\textbf{else}$
$\quad readfds'' = \text{INSERT\_ORDERED}\ \ readfds'\ readfds\ badreadfds \wedge$
$\quad writefds'' = \text{INSERT\_ORDERED}\ \ writefds'\ writefds\ badwritefds \wedge$
$\quad exceptfds'' = \text{INSERT\_ORDERED}\ \ exceptfds'\ exceptfds\ badexceptfds$

Description

Thread *tid* is blocked in state $\text{PSELECT2}(readfds, writefds, exceptfds)$. The call now returns three sets: $readfds''$, $writefds''$, and $exceptfds''$. $readfds''$ is the set of valid file descriptors in *readfds* that are ready for reading: a blocking $\mathsf{recv}(fd, \_, \_)$ call would not block; see soreadable (p202) for details. $writefds''$ is the set of valid file descriptors in *writefds* that are ready for writing: a blocking $\mathsf{send}(fd, \_, \_)$ call would not block; see sowriteable (p202) for details. $exceptfds''$ is the set of valid file descriptors in *exceptfds* that have pending exceptional conditions; see soexceptional (p203) for details.

Either one of these three sets is not empty or the timer d, which was set to the timeout value specified when the pselect() call was made, has expired.

A τ transition is made, leaving the thread state $\text{RET}(\text{OK}(readfds'', writefds'', exceptfds''))$.

Variations

FreeBSD	Invalid file descriptors (ones not in the host's finite map of file descriptors, *h.fds*) may be present in the sets *readfds*, *writefds*, and *exceptfds*, and all such file descriptors will then be included in the return sets $readfds''$, $writefds''$, and $exceptfds''$.

pselect_4 **all: fast fail** **Fail with EINVAL: Timeout not well-formed**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle$$

$$\xrightarrow{tid\cdot \mathsf{pselect}(readfds,\, writefds,\, exceptfds,\, timeout,\, sigmask)}$$

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched\_timer}})\rangle\!\rangle$$

$\neg(\text{tltimeopt\_wf}\ \ timeout)$

Description

From thread *tid*, which is in the RUN state, a pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) call is made. The *timeout* value is not well-formed: $timeout = \uparrow(s, ns)$ where either s is negative; ns is negative; or $ns > 1000000000$. The call fails with an EINVAL error.

A *tid*·pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) transition is made, leaving the thread state RET(FAIL EINVAL).

Model details

Such negative values are not admitted by the POSIX interface type but are by the model interface type (with (int ∗ int) option timeouts), so we check and generate EINVAL in the wrapper.

pselect_5 **all: fast fail** **Fail with EINVAL: File descriptor out of range**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle$$

$$\xrightarrow{tid\cdot \mathsf{pselect}(readfds,\, writefds,\, exceptfds,\, timeout,\, sigmask)}$$

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched\_timer}})\rangle\!\rangle$$

$$\begin{array}{l}
(\exists fd\ n.(fd\ \in\ readfds \lor fd\ \in\ writefds \lor fd\ \in\ exceptfds) \land \\
\quad\quad \textbf{if}\ \ \text{windows_arch}\ \ h.arch \\
\quad\quad\ \textbf{then}\ \ n = \textbf{max}(\textbf{length}\ \ readfds)(\textbf{max}(\textbf{length}\ \ writefds)(\textbf{length}\ \ exceptfds)) \land \\
\quad\quad\quad\quad n \geq \text{FD_SETSIZE}\ \ h.arch \\
\quad\quad\ \textbf{else} \\
\quad\quad fd = \text{FD}\ n \land \\
\quad\quad n \geq \text{FD_SETSIZE}\ \ h.arch) \lor \\
(\text{windows_arch}\ \ h.arch \land readfds = [\,] \land writefds = [\,] \land exceptfds = [\,])
\end{array}$$

Description

From thread *tid*, which is in the RUN state, a pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) call is made. One or more of the file descriptors in *readfds*, *writefds*, or *exceptfds* is greater than the architecure dependent FD_SETSIZE, the maximum file descriptor that can be specified in a pselect() call. The call fails with an EINVAL error.

A *tid*·pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

WinXP	On WinXP FD_SETSIZE is the maximum number of file descriptors in a set, so one of the sets *readfds*, *writefds*, or *exceptfds* has more than FD_SETSIZE members. Also, the call will fail with EINVAL if the sets *readfds*, *writefds*, and *exceptfds* are all empty.

pselect_6 **all: fast fail** **Fail with EBADF or ENOTSOCK: Bad file descriptor**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid\cdot \mathsf{pselect}(readfds,\ writefds,\ exceptfds,\ timeout,\ sigmask)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ err))_{\text{sched\_timer}})\rangle\!\rangle$$

$\neg\,\text{bsd\_arch}\ \ h.arch \land$
$(\exists fd.(fd\ \in\ readfds \lor fd\ \in\ writefds \lor fd\ \in\ exceptfds) \land$
$\quad fd\ \notin \mathbf{dom}(h.fds)) \land$
$(\mathbf{if}\ \ \text{windows\_arch}\ \ h.arch\ \ \mathbf{then}\ \ err = \text{ENOTSOCK}$
$\ \mathbf{else}\ \ err = \text{EBADF})$

Description

From thread *tid*, which is in the RUN state, a pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) call is made. There exists a file descriptor *fd* in *readfds*, *writefds*, or *exceptfds* that is not a valid file descriptor. The call fails with an EBADF error on FreeBSD and Linux and an ENOTSOCK error on WinXP.

A *tid*·pselect(*readfds*, *writefds*, *exceptfds*, *timeout*, *sigmask*) transition is made, leaving the thread state RET(FAIL *err*) where *err* is one of the above errors.

Variations

FreeBSD	This rule does not apply.
Linux	As above: the call fails with an EBADF error.
WinXP	As above: the call fails with an ENOTSOCK error.

15.19 recv() (TCP only)

recv : fd ∗ int ∗ msgbflag list → (string ∗ ((ip ∗ port) ∗ bool) option)

A call to recv(fd, *n*, *opts*) reads data from a socket's receive queue. This section describes the behaviour for TCP sockets. Here fd is a file descriptor referring to a TCP socket to read data from, *n* is the number of bytes of data to read, and *opts* is a list of message flags. Possible flags are:

- MSG_DONTWAIT: Do not block if there is no data available.
- MSG_OOB: Return out-of-band data.
- MSG_PEEK: Read data but do not remove it from the socket's receive queue.
- MSG_WAITALL: Block until all *n* bytes of data are available.

The returned string is the data read from the socket's receive queue. The ((ip ∗ port) ∗ bool) option is always returned as ∗ for a TCP socket.

In order to receive data, a TCP socket must be connected to a peer; otherwise, the recv() call will fail with an ENOTCONN error. If the socket has a pending error then the recv() call will fail with this error even if there is data available.

If there is no data available and non-blocking behaviour is not enabled (the socket's O_NONBLOCK flag is not set and the MSG_DONTWAIT flag was not used) then the recv() call will block until data arrives or an error occurs. If non-blocking behaviour is enabled and there is no data or error then the call will fail with an EAGAIN error.

The MSG_OOB flag can be set in order to receive out-of-band data; for this, the socket's SO_OOBINLINE cannot be set (i.e. out-of-band data must not be being returned inline).

15.19.1 Errors

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

EAGAIN	Non-blocking recv() call made and no data available; or out-of-band data requested and none is available.
EINVAL	Out-of-band data requested and SO_OOBINLINE flag set or the out-of-band data has already been read.
ENOTCONN	Socket not connected.
ENOTSOCK	The file descriptor passed does not refer to a socket.
EBADF	The file descriptor passed is not a valid file descriptor.
EINTR	The system was interrupted by a caught signal.
ENOBUFS	Out of resources.
ENOMEM	Out of resources.

15.19.2 Common cases

A TCP socket is created and then connected to a peer; a recv() call is made to receive data from that peer: *socket_1*; *return_1*; *connect_1*; *return_1*; *recv_1*; ...

15.19.3 API

Posix:	`ssize_t recv(int socket, void *buffer, size_t length, int flags);`
FreeBSD:	`ssize_t recv(int s, void *buf, size_t len, int flags);`
Linux:	`int recv(int s, void *buf, size_t len, int flags);`
WinXP:	`int recv(SOCKET s, char* buf, int len, int flags);`

In the Posix interface:

- `socket` is the file descriptor of the socket to receive from, corresponding to the fd argument of the model recv().
- `buffer` is a pointer to a buffer to place the received data in, which upon return contains the data received on the socket. This corresponds to the string return value of the model recv().
- `length` is the amount of data to be read from the socket, corresponding to the int argument of the model recv(); it should be at most the length of `buffer`.
- `flags` is a disjunction of the message flags that are set for the call, corresponding to the msgbflag list argument of the model recv().
- the returned `ssize_t` is either non-negative, in which case it is the the amount of data that was received by the socket, or it is `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

There are other functions used to receive data on a socket. `recvfrom()` is similar to `recv()` except it returns the source address of the data; this is used for UDP but is not necessary for TCP as the source address will always be the peer the socket has connected to. `recvmsg()`, another input function, is a more general form of `recv()`.

15.19.4 Model details

If the call blocks then the thread enters state $\textsc{Recv2}(\mathsf{sid}, n, opts)$ where:

- $\mathsf{sid} : \mathsf{sid}$ is the identifier of the socket that the recv() call was made on,
- $n : \mathsf{num}$ is the number of bytes to be read, and
- $opts : \mathsf{msgbflag\ list}$ is the list of message flags.

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, EFAULT can be returned if the `buffer` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model recv().
- In Posix, `EIO` may be returned to indicated that an I/O error occurred while reading from or writing to the file system; this is not modelled here.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

The following Linux message flags are not modelled: `MSG_NOSIGNAL`, `MSG_TRUNC`, and `MSG_ERRQUEUE`.

15.19.5 Summary

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_5	**tcp: fast succeed**	Successfully read non-inline out-of-band data
recv_6	**tcp: fast fail**	Fail with EAGAIN or EINVAL: recv() called with MSG_OOB set and out-of-band data is not available
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

15.19.6 Rules

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

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{tid\cdot \mathsf{recv}(fd, n_0, opts_0)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}(\mathbf{implode}\ str, *)))_{\mathrm{sched\_timer}});\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq'', rcvurp', iobc)))]|\!\rangle
\end{array}
$$

$((st \in \{\mathrm{ESTABLISHED}; \mathrm{FIN\_WAIT\_1}; \mathrm{FIN\_WAIT\_2}; \mathrm{CLOSING};$
$\mathrm{TIME\_WAIT}; \mathrm{CLOSE\_WAIT}; \mathrm{LAST\_ACK}\} \wedge$

$is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2) \vee$
$(st = \text{CLOSED})) \wedge$
$n = \text{clip\_int\_to\_num}\ \ n_0 \wedge$
$opts = \textbf{list\_to\_set}\ \ opts_0 \wedge$
$fd\ \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\text{MSG\_OOB}\ \notin\ opts \wedge$

(* 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. *)
$\textbf{let}\ \ have\_all\_data = (\textbf{length}\ \ rcvq \geq n)\ \textbf{in}$
$\textbf{let}\ \ have\_enough\_data = (\textbf{length}\ \ rcvq \geq sf.n(\text{SO\_RCVLOWAT}))\ \textbf{in}$
$\textbf{let}\ \ partial\_data\_ok = (\text{MSG\_WAITALL}\ \notin\ opts \vee n > sf.n(\text{SO\_RCVBUF}) \vee$
$\qquad (\neg(\text{bsd\_arch}\ \ h.arch) \wedge \text{MSG\_PEEK}\ \in\ opts))\ \textbf{in}$
$\textbf{let}\ \ urgent\_data\_ahead = (\exists om.rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \textbf{length}\ \ rcvq)\ \textbf{in}$
$(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee urgent\_data\_ahead \vee cantrcvmore) \wedge$

$((str, rcvq') = \text{SPLIT}(\textbf{min}\ \ n$
$\qquad (\textbf{case}\ \ rcvurp\ \ \textbf{of}$
$\qquad\quad * \rightarrow \textbf{length}\ \ rcvq\ \|$
$\qquad\quad \uparrow om \rightarrow \textbf{if}\ \ om = 0\ \ \textbf{then}\ (\textbf{length}\ \ rcvq)$
$\qquad\qquad \textbf{else}\ \ \textbf{min}\ \ om(\textbf{length}\ \ rcvq)))$
$\qquad rcvq) \wedge$
$rcvq'' = (\textbf{if}\ \ \text{MSG\_PEEK}\ \in\ opts\ \ \textbf{then}\ \ rcvq\ \ \textbf{else}\ \ rcvq') \wedge$
$rcvurp' = (\textbf{case}\ \ rcvurp\ \ \textbf{of}$
$\qquad * \rightarrow *\ \|$
$\qquad \uparrow om \rightarrow \textbf{if}\ \ om = 0\ \ \textbf{then}\ \ *$

$\qquad\qquad \textbf{else}\ \ \textbf{if}\ \ om \leq \textbf{length}\ \ str\ \ \textbf{then}\ \uparrow 0\ \ \textbf{else}\ \uparrow(om - \textbf{length}\ \ str))$

Description

From thread *tid*, which is in the RUN state, a $\text{recv}(fd, n_0, opts_0)$ 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, $sf.n(\text{SO\_RCVLOWAT})$, 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 $tid{\cdot}\text{recv}(fd, n_0, opts_0)$ transition is made leaving the thread state $\text{RET}(\text{OK}(\textbf{implode}\ \ str, *))$. 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**

$$h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (\text{Run})_d)\rbrack\!\rangle \xrightarrow{tid\cdot\mathsf{recv}(fd, n_0, opts_0)} h\ \langle\!\lbrack ts := ts \oplus (tid \mapsto (\text{Recv2}(sid, n, opts))_{\text{never\_timer}})\rbrack\!\rangle$$

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

(* We block if not enough (see *recv_1* (p209)) data is available and there is no pending error. *)
$\textbf{let}\ blocking = \neg(\text{MSG\_DONTWAIT}\ \in\ opts \vee ff.b(\text{O\_NONBLOCK}))\ \textbf{in}$
$\textbf{let}\ have\_all\_data = (\textbf{length}\ rcvq \geq n)\ \textbf{in}$
$\textbf{let}\ have\_enough\_data = (\textbf{length}\ rcvq \geq sf.n(\text{SO\_RCVLOWAT}))\ \textbf{in}$
$\textbf{let}\ partial\_data\_ok = (\text{MSG\_WAITALL}\ \notin\ opts \vee n > sf.n(\text{SO\_RCVBUF}) \vee$
$\qquad (\neg(\text{bsd\_arch}\ h.arch) \wedge \text{MSG\_PEEK}\ \in\ opts))\ \textbf{in}$
$\textbf{let}\ urgent\_data\_ahead = (\exists om.rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \textbf{length}\ rcvq)\ \textbf{in}$
$blocking\ \wedge$
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee urgent\_data\_ahead \vee cantrcvmore)\ \wedge$
$es = *$

Description

From thread *tid*, which is in the Run state, a $\mathsf{recv}(fd, n_0, opts_0)$ 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 $tid\cdot\mathsf{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{Recv2}(sid, n, opts)$.

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**

$$h \ \langle\!\![ts := ts \oplus (tid \mapsto (\mathrm{RECV2}(sid, n, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, \mathrm{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$$
$$\mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]]\!\!\rangle$$

$$\xrightarrow{\tau} h \ \langle\!\![ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ str, *)))_{\mathrm{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, \mathrm{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$$
$$\mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq'', rcvurp', iobc)))]]\!\!\rangle$$

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

(* We return at last if we now have enough (see *recv_1* (p209)) data available. Pending errors are not checked. *)

$$\mathbf{let}\ have\_all\_data = (\mathbf{length}\ rcvq \geq n)\ \mathbf{in}$$
$$\mathbf{let}\ have\_enough\_data = (\mathbf{length}\ rcvq \geq sf.n(\mathrm{SO\_RCVLOWAT}))\ \mathbf{in}$$
$$\mathbf{let}\ partial\_data\_ok = (\mathrm{MSG\_WAITALL} \notin opts \vee n > sf.n(\mathrm{SO\_RCVBUF}) \vee$$
$$(\neg(\mathrm{bsd\_arch}\ h.arch) \wedge \mathrm{MSG\_PEEK} \in opts))\ \mathbf{in}$$
$$\mathbf{let}\ urgent\_data\_ahead = (\exists om.rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \mathbf{length}\ rcvq)\ \mathbf{in}$$
$$(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee urgent\_data\_ahead \vee cantrcvmore) \wedge$$

$$(str, rcvq') = \mathrm{SPLIT}(\mathbf{min}\ n$$
$$(\mathbf{case}\ rcvurp\ \mathbf{of}$$
$$* \to \mathbf{length}\ rcvq\ \|$$
$$\uparrow om \to \mathbf{if}\ om = 0\ \mathbf{then}\ (\mathbf{length}\ rcvq)$$
$$\mathbf{else}\ \mathbf{min}\ om(\mathbf{length}\ rcvq)))$$
$$rcvq \wedge$$
$$rcvq'' = (\mathbf{if}\ \mathrm{MSG\_PEEK} \in opts\ \mathbf{then}\ rcvq\ \mathbf{else}\ rcvq') \wedge$$
$$rcvurp' = (\mathbf{case}\ rcvurp\ \mathbf{of}$$
$$* \to * \ \|$$
$$\uparrow om \to \mathbf{if}\ om = 0\ \mathbf{then}\ *$$
$$\mathbf{else}\ \mathbf{if}\ om \leq \mathbf{length}\ str\ \mathbf{then}\ \uparrow 0\ \mathbf{else}\ \uparrow(om - \mathbf{length}\ str))$$

Description

Thread *tid* is in the $\mathrm{RECV2}(sid, n, opts)$ 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, $sf.n(\mathrm{SO\_RCVLOWAT})$, 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 $\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ str, *))$. 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 $\uparrow(om - \textbf{length}\ str)$.

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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle$$
$$\xrightarrow{tid\cdot\mathsf{recv}(fd,\, n_0,\, opts_0)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched\_timer}})|\!\rangle$$

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

(* We fail if we would otherwise block (see *recv_2* (p211); these conditions are identical). *)
$\textbf{let}\ blocking = \neg(\text{MSG\_DONTWAIT}\ \in\ opts \vee ff.b(\text{O\_NONBLOCK}))\ \textbf{in}$
$\textbf{let}\ have\_all\_data = (\textbf{length}\ rcvq \geq n)\ \textbf{in}$
$\textbf{let}\ have\_enough\_data = (\textbf{length}\ rcvq \geq sf.n(\text{SO\_RCVLOWAT}))\ \textbf{in}$
$\textbf{let}\ partial\_data\_ok = (\text{MSG\_WAITALL}\ \notin\ opts \vee n > sf.n(\text{SO\_RCVBUF}) \vee$
$\qquad (\neg(\text{bsd\_arch}\ h.arch) \wedge \text{MSG\_PEEK}\ \in\ opts))\ \textbf{in}$
$\textbf{let}\ urgent\_data\_ahead = (\exists om.rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \textbf{length}\ rcvq)\ \textbf{in}$
$\neg blocking\ \wedge$
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee urgent\_data\_ahead \vee cantrcvmore)\ \wedge$
$(rcvq = [\,] \implies es = *)$

Description

From thead *tid*, which is in the RUN state, a $\mathsf{recv}(fd, n_0, opts_0)$ 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 $tid\cdot\mathsf{recv}(fd, n_0, opts_0)$ 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_5 **tcp: fast succeed** **Successfully read non-inline out-of-band data**

$$h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\ socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore, \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)} h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\textbf{implode}\ str, *)))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore, \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc')))]|\!\rangle$$

$n = \text{clip\_int\_to\_num}\ n_0 \wedge$
$opts = \textbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$\text{MSG\_OOB} \in opts \wedge$
$\neg sf.b(\text{SO\_OOBINLINE}) \wedge$
$iobc = \text{OOBDATA}\ c \wedge$
$str = (\textbf{if}\ n = 0\ \textbf{then}\ [\,]\ \textbf{else}\ [c]) \wedge$
$iobc' = (\textbf{if}\ \text{MSG\_PEEK} \in opts\ \textbf{then}\ iobc\ \textbf{else}\ \text{HAD\_OOBDATA})$

Description

From thread tid, which is in the RUN state, a $\mathsf{recv}(fd, n_0, opts_0)$ call is made. fd refers to a TCP socket sid with binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$ and no pending error. Out-of-band data is requested: the MSG_OOB flag is set in $opts_0$, and out-of-band data is not being returned inline: $\neg sf.b(\text{SO\_OOBINLINE})$. There is a byte c of out-of-band data on the socket; if zero bytes of data were requested, $n_0 = 0$, then the empty string is returned, otherwise c is returned.

A $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(\textbf{implode}\ str, *))$ where **implode** *str* is the returned out-of-band data. If the MSG_PEEK flag was set in $opts_0$ then the byte of out-of-band data is left in place, $iobc' = iobc$; otherwise it is removed and marked as read: $iobc' = \text{HAD\_OOBDATA}$.

Model details

The amount of data requested, n_0, is clipped to a natural number from an integer, using clip_int_to_num. POSIX specifies an unsigned type for n_0, whereas the model uses int.

The $opts_0$ argument to recv() is of type msgbflag list, but it is converted to a set, *opts*, using **list_to_set**.

The data itself is represented as a *byte* list in the datagram but is returned a string: the **implode** function is used to do the conversion.

recv_6 **tcp: fast fail** **Fail with EAGAIN or EINVAL:** recv() **called with MSG_OOB set and out-of-band data is not available**

$$h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle \xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)} h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ e))_{\text{sched\_timer}})|\!\rangle$$

$n = \text{clip\_int\_to\_num}\ n_0 \wedge$
$opts = \textbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \textbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$

$h.socks[sid] = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore,$
$\text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$\text{MSG\_OOB} \in opts \wedge$
$(\textbf{if } sf.b(\text{SO\_OOBINLINE})$
$\textbf{ then } (e = \text{EINVAL})$
$\textbf{ else case } iobc \textbf{ of}$

$\text{NO\_OOBDATA} \rightarrow (e = \textbf{if } rcvurp = * \textbf{ then } \text{EINVAL } \textbf{else } \text{EAGAIN}) \parallel$
$\text{OOBDATA } c \rightarrow \mathbf{F} \parallel$
$\text{HAD\_OOBDATA} \rightarrow (e = \text{EINVAL}))$

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 identified by *sid* with binding quad ($\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2$) and no pending error. The MSG_OOB flag is set in $opts_0$, indicating that out-of-band data should be returned, but no out-of-band data is available because either: (1) out-of-band data is being returned in-line (the $sf.b$(SO_OOBINLINE) flag is set); (2) the out-of-band data on the socket has already been read; (3) there is no out-of-band data and the receive urgent pointer is set; or (4) there is no out-of-band data but the urgent pointer is set, corresponding to the case where the peer has advertised urgent data but that data has yet to arrive. The call fails with an EINVAL error in cases (1), (2), and (3); and a EAGAIN error in case (4) indicating that the recv() call should be made again to see if the data has now arrived.

A $tid \cdot$recv($fd, n_0, opts_0$) transition is made, leaving the thread state Ret(FAIL e) where e is one of the above errors.

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

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

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

)

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**

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

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

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

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

(* We fail immediately if there is a pending error and we could not otherwise return data (see *recv_1* (p209)). *)
let $rcvq = tcp\_sock.rcvq$ **in**
let $rcvurp = tcp\_sock.rcvurp$ **in**
let $blocking = \neg(\text{MSG\_DONTWAIT} \in opts \vee ff.b(\text{O\_NONBLOCK}))$ **in**
let $have\_all\_data = (\textbf{length } rcvq \geq n)$ **in**
let $have\_enough\_data = (\textbf{length } rcvq \geq sf.n(\text{SO\_RCVLOWAT}))$ **in**
let $partial\_data\_ok = (\text{MSG\_WAITALL} \notin opts \vee n > sf.n(\text{SO\_RCVBUF}) \vee$
$(\neg(\text{bsd\_arch } h.arch) \wedge \text{MSG\_PEEK} \in opts))$ **in**
let $urgent\_data\_ahead = (\exists om. rcvurp = \uparrow om \wedge 0 < om \wedge om \leq \textbf{length } rcvq)$ **in**
$\neg(have\_all\_data \vee (have\_enough\_data \wedge partial\_data\_ok) \vee urgent\_data\_ahead) \wedge$
$(blocking \vee rcvq = [\,]) \wedge$

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

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 ⦇$ts := ts \oplus (tid \mapsto (\text{RECV2}(sid, n, opts))_d)$;
$socks := socks \oplus$
$[(sid, sock$ ⦇$es := \uparrow e; pr := \text{TCP\_PROTO}(tcp\_sock)$⦈$)]$⦈

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

(* We fail now if there is a pending error and we could not otherwise return data (see *recv_1* (p209)). *)
let $have\_all\_data = (\textbf{length}\ tcp\_sock.rcvq \geq n)$ **in**
let $have\_enough\_data = (\textbf{length}\ tcp\_sock.rcvq \geq sock.sf.n(\text{SO\_RCVLOWAT}))$ **in**
let $partial\_data\_ok = (\text{MSG\_WAITALL} \notin opts \lor n > sock.sf.n(\text{SO\_RCVBUF}) \lor$
$(\neg(\text{bsd\_arch}\ h.arch) \land \text{MSG\_PEEK} \in opts))$ **in**
let $urgent\_data\_ahead = (\exists om.tcp\_sock.rcvurp = \uparrow om \land 0 < om \land om \leq \textbf{length}\ tcp\_sock.rcvq)$ **in**
$\neg(have\_all\_data \lor (have\_enough\_data \land partial\_data\_ok) \lor urgent\_data\_ahead) \land$

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

Description

Thread *tid* is blocked in state $\text{RECV2}(sid, n, opts)$ 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 $\text{RET}(\text{FAIL}\ e)$. 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**

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

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

Description

On WinXP, from thread *tid*, which is in the RUN state, a $\text{recv}(fd, n, opts)$ 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 $tid \cdot \text{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL ESHUTDOWN})$.

Variations

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

15.20 recv() (UDP only)

recv : (fd ∗ int ∗ msgbflag list) → (string ∗ ((ip ∗ port) ∗ bool) option)

A call to recv(fd, n, *opts*) returns data from the datagram on the head of a socket's receive queue. This section describes the behaviour for UDP sockets. Here the fd argument is a file descriptor referring to the socket to receive data from, n specifies the number of bytes of data to read from that socket, and the *opts* argument is a list of flags for the recv() call. The possible flags are:

- MSG_DONTWAIT: non-blocking behaviour is requested for this call. This flag only has effect on Linux. FreeBSD and WinXP ignore it. See rules *recv_12* and *recv_13*.
- MSG_PEEK: return data from the datagram on the head of the receive queue, without removing that datagram from the receive queue.
- MSG_WAITALL: do not return until all n bytes of data have been read. Linux and FreeBSD ignore this flag. WinXP fails with EOPNOTSUPP as this is not meaningful for UDP sockets: the returned data is from only one datagram.
- MSG_OOB: return out-of-band data. This flag is ignored on Linux. On WinXP and FreeBSD the call fails with EOPNOTSUPP as out-of-band data is not meaningful for UDP sockets.

The returned value of the recv() call, (string ∗ ((ip ∗ port) ∗ bool) option), consists of the data read from the socket (the string), the source address of the data (the ip ∗ port), and a flag specifying whether or not all of the datagram's data was read (the bool). The latter two components are wrapped in an option type (for type compatibility with the TCP recv()) but are always returned for UDP. The flag only has meaning on WinXP and should be ignored on FreeBSD and Linux.

For a socket to receive data, it must be bound to a local port. On Linux and FreeBSD, if the socket is not bound to a local port, then it is autobound to an ephemeral port when the recv() call is made. On WinXP, calling recv() on a socket that is not bound to a local port is an EINVAL error.

If a non-blocking recv() call is made (the socket's O_NONBLOCK flag is set) and there are no datagrams on the socket's receive queue, then the call will fail with EAGAIN. If the call is a blocking one and the socket's receive queue is empty then the call will block, returning when a datagram arrives or an error occurs.

If the socket has a pending error then on FreeBSD and Linux, the call will fail with that error. On WinXP, errors from ICMP messages are placed on the socket's receive queue, and so the error will only be returned when that message is at the head of the receive queue.

15.20.1 Errors

A call to recv() can fail with the errors below, in which case the corresponding exception is raised.

EAGAIN	The call would block and non-blocking behaviour is requested. This is done either via the 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.

15.20.2 Common cases

A UDP socket is created and bound to a local address. Other calls are made and datagrams are delivered to the socket; recv() is called to read from a datagram: *socket_1*; *return_1*; *bind_1*; ... *recv_11*; *return_1*;

A UDP socket is created and bound to a local address. recv() is called and blocks; a datagram arrives addressed to the socket's local address and is placed on its receive queue; the call returns: *socket_1*; *return_1*; *bind_1*; ... *recv_12*; *deliver_in_99*; *deliver_in_udp_1*; *recv_15*; *return_1*;

15.20.3 API

Posix:	`ssize_t recvfrom(int socket, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict address, socklen_t *restrict address_len);`
FreeBSD:	`ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);`
Linux:	`int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);`
WinXP:	`int recvfrom(SOCKET s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen);`

In the Posix interface:

- `socket` is the file descriptor of the socket to receive from, corresponding to the fd argument of the model recv().
- `buffer` is a pointer to a buffer to place the received data in, which upon return contains the data received on the socket. This corresponds to the string return value of the model recv().
- `length` is the amount of data to be read from the socket, corresponding to the int argument of the model recv(); it should be at most the length of `buffer`.
- `flags` is a disjunction of the message flags that are set for the call, corresponding to the 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()`.

15.20.4 Model details

If the call blocks then the thread enters state RECV2(sid, n, *opts*) where:

- sid : sid is the identifier of the socket that the recv() call was made on,
- n : num is the number of bytes to be read, and
- *opts* : msgbflag list is the set of message flags.

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, EFAULT can be returned if the `buffer` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model recv().
- In Posix, `EIO` may be returned to indicated that an I/O error occurred while reading from or writing to the file system; this is not modelled here.
- `EINVAL` may be returned if the MSG_OOB flag is set and no out-of-band data is available; out-of-band data does not exist for UDP so this does not apply.
- `ENOTCONN` may be returned if the socket is not connected; this does not apply for UDP as the socket need not have a peer specified to receive datagrams.
- `ETIMEDOUT` can be returned due to a transmission timeout on a connection; UDP is not connection-oriented so this does not apply.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

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

15.20.5 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

15.20.6 Rules

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

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

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

$$\begin{array}{l}
fd \ \in \mathbf{dom}(h.fds) \wedge \\
\quad fid = h.fds[fd] \wedge \\
\quad h.files[fid] = \mathrm{FILE}(\mathrm{FT_SOCKET}(sid), ff) \wedge \\
\quad sock = \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, cantsndmore, cantrcvmore, \mathrm{UDP_Sock}(rcvq')) \wedge \\
\quad (\neg(\mathrm{linux_arch}\ \ h.arch) \implies cantrcvmore = \mathbf{F}) \wedge \\
\quad rcvq = (\mathrm{DGRAM_MSG}(\langle\!|\ is := i_3; ps := ps_3; data := data|\!\rangle)) :: rcvq'' \wedge \\
\quad n = \mathrm{clip_int_to_num}\ \ n_0 \wedge \\
\quad ((\mathbf{length}\ \ data \leq n \wedge data = data') \vee \\
\quad\quad (\mathbf{length}\ \ data > n \wedge data' = \mathrm{TAKE}\ \ n\ data \wedge \mathbf{length}\ \ data' = n \wedge \neg(\mathrm{windows_arch}\ \ h.arch))) \wedge \\
\quad (\mathrm{windows_arch}\ \ h.arch \implies b = \mathbf{T}) \wedge \\
\quad opts = \mathbf{list_to_set}\ \ opts_0 \wedge \\
\quad rcvq' = (\mathbf{if}\ \ \mathrm{MSG_PEEK}\ \in\ opts\ \ \mathbf{then}\ \ rcvq\ \ \mathbf{else}\ \ rcvq'')
\end{array}$$

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 $\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ \ data', \uparrow(i_3, ps_3)))$, 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).

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**

$$h_0 \xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)} h_0 \, \langle\!\lbrack ts := ts \oplus (tid \mapsto (\text{RECV2}(sid, n, opts))_{\text{never\_timer}}); \; socks := h_0.socks \oplus [(sid, sock \, \langle\!\lbrack ps_1 := \uparrow p_1' \rbrack\!\rangle)]; \; bound := bound \rbrack\!\rangle$$

$h_0 = h \, \langle\!\lbrack \, ts := ts \oplus (tid \mapsto (\text{RUN})_d);$
$\quad socks := socks \oplus$
$\quad [(sid, sock)] \rbrack\!\rangle \wedge$
$fd \in \mathbf{dom}(h_0.fds) \wedge$
$fid = h_0.fds[fd] \wedge$
$h_0.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, *, cantsndmore, \mathbf{F}, \text{UDP\_Sock}([\,])) \wedge$
$p_1' \in \text{autobind}(sock.ps_1, \text{PROTO\_UDP}, h_0.socks) \wedge$
$(\mathbf{if}\ sock.ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \wedge$
$\neg((\text{MSG\_DONTWAIT} \in opts \wedge \text{linux\_arch}\ h.arch) \vee ff.b(\text{O\_NONBLOCK})) \wedge$
$(\text{bsd\_arch}\ h.arch \implies \neg(n = 0)) \wedge$
$n = \text{clip\_int\_to\_num}\ n_0 \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0$

Description

Consider a UDP socket *sid*, referenced by *fd*, that has no pending errors, is not shutdown for reading, has an empty receive queue, and does not have its O_NONBLOCK flag set. The socket is either bound to a local port $\uparrow p_1'$ or can be autobound to a local port $\uparrow p_1'$. From thread *tid*, which in the RUN state, a $\mathsf{recv}(fd, n_0, opts_0)$ call is made. Because there are no datagrams on the socket's receive queue, the call will block.

A $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition will be made, leaving the thread state $\text{RECV2}(sid, n, opts)$. 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

$$h_0 \xrightarrow{tid\cdot\mathsf{recv}(fd,\,n,\,opts_0)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL\ EAGAIN}))_{\mathrm{sched\_timer}});\ socks := socks \oplus [(sid, s\ \langle\!\langle es := *; pr := \mathrm{UDP\_Sock}([\,])\rangle\!\rangle)]\rangle\!\rangle$$

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

Description

Consider a UDP socket *sid* referenced by *fd*. It has no pending errors, and an empty receive queue. The socket is non-blocking: its O_NONBLOCK flag has been set. From thread *tid*, in the RUN state, a $\mathsf{recv}(fd, n, opts_0)$ call is made. The call would block because the socket has an empty receive queue, so the call fails with an EAGAIN error.

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

Model details

The $opts_0$ argument is of type list. In the model it is converted to a set *opts* using **list_to_set**.

Variations

Linux	As above, but the rule also applies if the socket's 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**

$$h_0 \xrightarrow{tid\cdot\mathsf{recv}(fd,\,n,\,opts)} h_0\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

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

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 $\mathsf{recv}(fd, n, opts)$ call is made. There is no ephemeral port to autobind the socket to, so the call fails with either EAGAIN, EADDRNOTAVAIL or ENOBUFS.

A $tid\cdot\mathsf{recv}(fd, n, opts)$ transition is made, leaving the thread state RET(FAIL e) where e is one of the above errors.

recv_15 **udp: slow urgent succeed** **Blocked call returns from Recv2 state with data**

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RECV2}(sid, n, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, sock \; \langle\!| ps_1 := \uparrow p_1; es := *; pr := \mathrm{UDP\_Sock}(rcvq) |\!\rangle)] |\!\rangle$$
$$\xrightarrow{\tau} \; h \; \langle\!| ts := ts \oplus (tid \mapsto \big(\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ data', \uparrow((i_3, ps_3), b)))\big)_{\mathrm{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, sock \; \langle\!| ps_1 := \uparrow p_1; es := *; pr := \mathrm{UDP\_Sock}(rcvq') |\!\rangle)] |\!\rangle$$

$rcvq = (\mathrm{DGRAM\_MSG}(\langle\!| \; is := i_3; ps := ps_3; data := data |\!\rangle)) :: rcvq'' \wedge$
$(rcvq' = \mathbf{if}\ \mathrm{MSG\_PEEK} \in opts\ \mathbf{then}\ rcvq\ \mathbf{else}\ rcvq'') \wedge$
$((\mathbf{length}\ data \leq n \wedge data = data') \vee$
$(\mathbf{length}\ data > n \wedge \neg(\mathrm{windows\_arch}\ h.arch) \wedge data' = \mathrm{TAKE}\ n\ data' \wedge \mathbf{length}\ data' = n)) \wedge$
$(\mathrm{windows\_arch}\ h.arch \implies b = \mathbf{T})$

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 $\mathrm{RECV2}(sid, n, opts)$.

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 $\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\, data', \uparrow((i_3, ps_3), b)))$. 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**

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);$$
$$socks := socks \oplus$$
$$[(sid, sock \; \langle\!| pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle$$
$$\xrightarrow{tid \cdot \mathrm{recv}(fd, n_0, opts_0)} \; h \; \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL\ EOPNOTSUPP}))_{\mathrm{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, sock \; \langle\!| pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$((\mathrm{MSG\_OOB} \in opts \wedge \neg(\mathrm{linux\_arch}\ h.arch)) \vee (\mathrm{MSG\_WAITALL} \in opts \wedge \mathrm{windows\_arch}\ h.arch))$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, in the RUN state, a $\mathrm{recv}(fd, n_0, opts_0)$ call is made. The MSG_OOB or MSG_WAITALL flags are set in $opts_0$. The call fails with an EOPNOTSUPP error.

A $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ 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**

$$
\begin{array}{l}
h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, sock\ \langle\!\langle cantrcvmore := \mathbf{T}; pr := \mathrm{UDP\_Sock}(rcvq)\rangle\!\rangle)]\rangle\!\rangle\\
\xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)}\ h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(ret))_{\mathrm{sched\_timer}});\\
\qquad socks := socks \oplus\\
\qquad\quad [(sid, sock\ \langle\!\langle cantrcvmore := \mathbf{T}; pr := \mathrm{UDP\_Sock}(rcvq)\rangle\!\rangle)]\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge\\
fid = h.fds[fd] \wedge\\
h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge\\
\mathbf{if}\ \ \mathrm{windows\_arch}\ h.arch\ \ \mathbf{then}\ \ ret = \mathrm{FAIL}\ (\mathrm{ESHUTDOWN}) \wedge rc = \text{FAST FAIL}\\
\ \mathbf{else\ if}\ \ \mathrm{bsd\_arch}\ h.arch\ \ \mathbf{then}\ \ ret = \mathrm{OK}(``", \uparrow((*, *), b)) \wedge rc = \text{FAST SUCCEED} \wedge\\
sock.es = *\\
\ \mathbf{else\ if}\ \ \mathrm{linux\_arch}\ h.arch\ \ \mathbf{then}\\
\quad rcvq = [\,] \wedge ret = \mathrm{OK}(``", \uparrow((*, *), b)) \wedge rc = \text{FAST SUCCEED} \wedge sock.es = *\\
\ \mathbf{else}\ \ \mathit{ASSERTION\_FAILURE}``\mathrm{recv\_17}"
\end{array}
$$

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 $\mathsf{recv}(fd, n_0, opts_0)$ call is made. On FreeBSD and Linux, if the socket has no pending error the call is successfully, returning $(``", \uparrow((*, *), b))$; on WinXP the call fails with an ESHUTDOWN error.

A $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\mathrm{RET}(\mathrm{OK}(``", \uparrow((*, *), b)))$ on FreeBSD and Linux, or RET(FAIL ESHUTDOWN) on WinXP.

Variations

FreeBSD	As above: the call succeeds.
Linux	As above: the call succeeds with the additional condition that the socket has an empty receive queue.
WinXP	As above: the call fails with an ESHUTDOWN error.

recv_20 **udp: rc Successful partial read of datagram on head of socket's receive queue on WinXP**

$$
\begin{array}{l}
h\ \langle\!\langle ts := ts \oplus (tid \mapsto (t)_d); \\
\quad socks := socks \oplus \\
\qquad [(sid, sock\ \langle\!\langle pr := \mathrm{UDP\_Sock}(rcvq)\rangle\!\rangle)]\rangle\!\rangle \\
\xrightarrow{lbl}\ h\ \langle\!\langle ts := ts \oplus (tid \mapsto \left(\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ \ data', \uparrow((i_3, ps_3), \mathbf{F})))\right)_{\mathrm{sched\_timer}}); \\
\quad socks := socks \oplus \\
\qquad [(sid, sock)]\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathrm{windows\_arch}\ \ h.arch \wedge \\
rcvq = (\mathrm{DGRAM\_MSG}(\langle\!\langle\ is := i_3; ps := ps_3; data := data\rangle\!\rangle)) :: rcvq'' \wedge \\
sock = \mathrm{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, *, cantsndmore, cantrcvmore, \mathrm{UDP\_Sock}(rcvq')) \wedge \\
((\exists fd\ ff\ \ n\ n_0\ opts_0. \\
\quad fd\ \in \mathbf{dom}(h.fds) \wedge \\
\quad fid = h.fds[fd] \wedge \\
\quad h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge \\
\quad (rcvq' = \mathbf{if}\ \ \mathrm{MSG\_PEEK}\ \in (\mathbf{list\_to\_set}\ \ opts_0)\ \mathbf{then}\ \ rcvq\ \ \mathbf{else}\ \ rcvq'') \wedge \\
\quad n = \mathrm{clip\_int\_to\_num}\ \ n_0 \wedge \\
\quad n < \mathbf{length}\ \ data \wedge \\
\quad data' = \mathrm{TAKE}\ \ n\ data \wedge \\
\quad t = \mathrm{RUN} \wedge \\
\quad rc = \mathrm{FAST\ SUCCEED} \wedge \\
\quad lbl = tid \cdot \mathsf{recv}(fd, n_0, opts_0)) \vee \\
(\exists n\ opts. \\
\quad lbl = \tau \wedge \\
\quad t = \mathrm{RECV2}(sid, n, opts) \wedge \\
\quad rc = \mathrm{SLOW}\ \ \mathrm{urgent}\ \ \mathrm{SUCCEED} \wedge \\
\quad data' = \mathrm{TAKE}\ \ n\ data \wedge \\
\quad n < \mathbf{length}\ \ data \wedge \\
\quad rcvq' = \mathbf{if}\ \ \mathrm{MSG\_PEEK}\ \in\ \ opts\ \ \mathbf{then}\ \ rcvq\ \ \mathbf{else}\ \ rcvq''))
\end{array}
$$

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 $\mathsf{recv}(fd, n_0, opts_0)$ 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 $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition is made leaving the thread state $\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\ \ data', \uparrow((i_3, ps_3), \mathbf{F})))$ 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 $\mathrm{RECV2}(sid, n, opts)$ 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 $\mathrm{RET}(\mathrm{OK}(\mathbf{implode}\, data', \uparrow((i_3, ps_3), \mathbf{F})))$ 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 $\mathrm{OK}(\mathbf{implode}\ \ data', \uparrow((i_3, p_3), \mathbf{F}))$ 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**

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

$$\xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)} h \; \langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(``", \uparrow((*,*), b))))_{\text{sched\_timer}}); \; socks := socks \oplus [(sid, sock \; \langle\!| pr := \text{UDP\_Sock}([\,]) |\!\rangle)] |\!\rangle$$

bsd_arch $h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$0 = \text{clip\_int\_to\_num} \; n_0$

Description

On FreeBSD, consider a UDP socket *sid*, referenced by *fd*, with an empty receive queue. From thread *tid*, which is in the RUN state, a $\mathsf{recv}(fd, n_0, opts_0)$ call is made where $n_0 = 0$. The call succeeds, returning the empty string and not specifying an address: $\text{OK}(``", \uparrow((*,*), b))$.

A $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition is made, leaving the thread state $\text{RET}(\text{OK}(``", \uparrow((*,*), b)))$.

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**

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d); \; socks := socks \oplus [(sid, sock \; \langle\!| ps_1 := *; pr := \text{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{recv}(fd, n_0, opts_0)} h \; \langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{sched\_timer}}); \; socks := socks \oplus [(sid, sock \; \langle\!| ps_1 := *; pr := \text{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle$$

windows_arch $h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$

$h.files[fid] = \textsc{File}(\textsc{FT\_Socket}(sid), ff)$

Description

On WinXP, consider a UDP socket *sid* referenced by *fd* that is not bound to a local port. A $\mathsf{recv}(fd, n_0, opts_0$ call is made from thread *tid* which is in the $\textsc{Run}$ state. The call fails with an EINVAL error.

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

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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d);\ socks := socks \oplus [(sid, sock\ \langle\!|pr := \text{UDP\_Sock}(rcvq)|\!\rangle)]|\!\rangle \xrightarrow{lbl} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } err))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!|pr := \text{UDP\_Sock}(rcvq')|\!\rangle)]|\!\rangle$$

$$\begin{aligned}
&\text{windows_arch}\ \ h.arch \wedge \\
&rcvq = (\textsc{Dgram_error}(\langle\!|\ e := err|\!\rangle)) :: rcvq' \wedge \\
&((\exists fd\ n_0\ opts_0\ fid\ ff.t = \textsc{Run} \wedge \\
&\qquad lbl = tid \cdot \mathsf{recv}(fd, n_0, opts_0) \wedge \\
&\qquad rc = \textsc{fast fail} \wedge \\
&\qquad fd\ \in \mathbf{dom}(h.fds) \wedge \\
&\qquad fid = h.fds[fd] \wedge \\
&\qquad h.files[fid] = \textsc{File}(\textsc{FT_Socket}(sid), ff)) \vee \\
&(\exists n\ opts.t = \textsc{Recv2}(sid, n, opts) \wedge \\
&\qquad lbl = \tau \wedge \\
&\qquad rc = \textsc{slow}\ \ \text{urgent}\ \ \textsc{fail}))
\end{aligned}$$

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 $\textsc{Run}$ state and a $\mathsf{recv}(fd, n_0, opts_0)$ call is made. The call fails with error *err*, making a $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ transition. This leaves the thread state $\textsc{Ret}(\text{FAIL } err)$, and the socket with the ICMP message removed from its receive queue.

In the second case, thread *tid* is blocked in state $\textsc{Recv2}(sid, n_0, opts_0)$. A τ transition is made, leaving the thread state $\textsc{Ret}(\text{FAIL } err)$, and the socket with the ICMP message removed from its receive queue.

Variations

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

recv_24 **udp: fast fail** **Fail with pending error**

h ⟨[$ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$;
$socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, \uparrow e, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]$]⟩

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

h ⟨[$ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{sched\_timer}})$;
$socks := socks \oplus$
$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))]$]⟩

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$(\neg\, \text{linux\_arch}\ h.arch \implies \exists p_2.ps_2 = \uparrow p_2) \wedge$
$es = \mathbf{if}\ \text{MSG\_PEEK} \in opts\ \mathbf{then} \uparrow e\ \mathbf{else}\ *$

Description

From thread *tid*, which is in the Run state, a $\mathsf{recv}(fd, n_0, opts_0)$ 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 $tid \cdot \mathsf{recv}(fd, n_0, opts_0)$ 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.

15.21 send() (TCP only)

send : fd ∗ (ip ∗ port) option ∗ string ∗ msgbflag list → string

This section describes the behaviour of send() for TCP sockets. A call to send(fd, ∗, *data*, *flags*) enqueues data on the TCP socket's send queue. Here fd is a file descriptor referring to the TCP socket to enqueue data on. The second argument, of type (ip ∗ port) option, is the destination address of the data for UDP, but for a TCP socket it should be set to ∗ (the socket must be connected to a peer before send() can be called). The *data* is the data to be sent. Finally, *flags* is a list of flags for the send() call; possible flags are: MSG_OOB, specifying that the data to be sent is out-of-band data, and MSG_DONTWAIT, specifying that non-blocking behaviour is to be used for this call. The MSG_WAITALL and MSG_PEEK flags may also be set, but as they are meaningless for send() calls, FreeBSD ignores them, and Linux and WinXP fail with EOPNOTSUPP. The returned string is any data that was not sent.

For a successful send() call, the socket must be in a synchronised state, must not be shutdown for writing, and must not have a pending error.

If there is not enough room on a socket's send queue then a send() call may block until space becomes available. For a successful blocking send() call on FreeBSD the entire string will be enqueued on the socket's send queue.

15.21.1 Errors

In addition to errors returned via ICMP (see *deliver_in_icmp_3* (p337)), a call to send() can fail with the errors below, in which case the corresponding exception is raised:

EAGAIN	Non-blocking send() call would block.
ENOTCONN	Socket not connected on FreeBSD and WinXP.
EOPNOTSUPP	Message flags MSG_PEEK and MSG_WAITALL not supported. Linux and WinXP.
EPIPE	Socket not connected on Linux; or socket shutdown for writing on FreeBSD and Linux.
ESHUTDOWN	Socket shutdown for writing on WinXP.
EBADF	The file descriptor passed is not a valid file descriptor.
EINTR	The system was interrupted by a caught signal.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.21.2 Common cases

A TCP socket is created and successfully connects with a peer; data is then sent to the peer: *socket_1*; *return_1*; *connect_1*; *return_1*; ... *connect_2*; *return_1*; *send_1*; ...

15.21.3 API

Posix:	`ssize_t send(int socket, const void *buffer, size_t length, int flags);`
FreeBSD:	`ssize_t send(int s, const void *msg, size_t len, int flags);`
Linux:	`int send(int s, const void *msg, size_t len, int flags);`
WinXP:	`int send(SOCKET s, const char *buf, int len, int flags);`

In the Posix interface:

- `socket` is the file descriptor of the socket to send from, corresponding to the fd argument of the model send().
- `message` is a pointer to the data to be sent of length `length`. The two together correspond to the string argument of the model send().
- `flags` is a disjunction of the message flags for the send() call, corresponding to the msgbflag list in the model send().
- the returned `ssize_t` is either non-negative or `-1`. If it is non-negative then it is the amount of data from `message` that was sent. If it is `-1` then it indicates an error, in which case the error is stored in `errno`. This corresponds to the model send()'s return value of type string which is the data that was not sent. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.21.4 Model details

If the call blocks then the thread enters state $\text{S}\small{\text{END}}2(\mathsf{sid}, *, \mathit{str}, \mathit{opts})$ (the optional parameter is used for UDP only), where

- sid : sid is the identifier of the socket that made the send() call,
- *str* : string is the data to be sent, and

- *opts* : msgbflag list is the set of options for the send() call.

The following errors are not modelled:

- In Posix and on all three architectures, `EDESTADDRREQ` indicates that the socket is not connection-mode and no peer address is set. This doesn't apply to TCP, which is a connection-mode protocol.
- In Posix, `EACCES` signifies that write access to the socket is denied. This is not modelled here.
- On FreeBSD and Linux, `EFAULT` signifies that the pointers passed as either the `address` or `address_len` arguments were inaccessible. This is an artefact of the C interface to accept() that is excluded by the clean interface used in the model.
- In Posix and on Linux, `EINVAL` signifies that an invalid argument was passed. The typing of the model interface prevents this from happening.
- In Posix, `EIO` signifies that an I/O error occurred while reading from or writing to the file system. This is not modelled.
- On Linux, `EMSGSIZE` indicates that the message is too large to be sent all at once, as the socket requires; this is not a requirement for TCP sockets.
- In Posix, `ENETDOWN` signifies that the local network interface used to reach the destination is down. This is not modelled.

The following flags are not modelled:

- On Linux, `MSG_CONFIRM` is used to tell the link layer not to probe the neighbour.
- On Linux, `MSG_NOSIGNAL` requests not to send `SIGPIPE` errors on stream-oriented sockets when the other end breaks the connection.
- On FreeBSD and WinXP, `MSG_DONTROUTE` is used by routing programs.
- On FreeBSD, `MSG_EOR` is used to indicate the end of a record for protocols that support this. It is not modelled because TCP does not support records.
- On FreeBSD, `MSG_EOF` is used to implement Transaction TCP which is not modelled here.

15.21.5 Summary

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

15.21.6 Rules

send_1 **tcp: fast succeed** **Successfully send data without blocking**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle$$

$$\xrightarrow{tid\cdot \mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)}$$

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

$st \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$

$space \in \text{send\_queue\_space}$
$\quad (sf.n(\text{SO\_SNDBUF}))(\mathbf{length}\ sndq)(\text{MSG\_OOB} \in opts)h.arch\ cb.t\_maxseg\ i_2 \wedge$

$(\{\text{MSG\_PEEK}; \text{MSG\_WAITALL}\} \cap opts = \emptyset \vee \text{bsd\_arch}\ h.arch) \wedge$

$(\mathbf{if}\ space \geq \mathbf{length}\ str\ \mathbf{then}$
$\quad str' = str \wedge str'' = [\,]$
$\mathbf{else}$
$\quad (ff.b(\text{O\_NONBLOCK}) \vee (\text{MSG\_DONTWAIT} \in opts \wedge \neg \text{bsd\_arch}\ h.arch)) \wedge$
$\quad (\mathbf{if}\ \text{bsd\_arch}\ h.arch\ \mathbf{then}\ space \geq sf.n(\text{SO\_SNDLOWAT})$
$\quad \mathbf{else}\ space > 0) \wedge$
$\quad (str', str'') = \text{SPLIT}\ space\ str$
$) \wedge$
$sndurp' = (\mathbf{if}\ (\text{MSG\_OOB} \in opts) \wedge (n = \mathbf{length}\ str)$
$\quad\quad \mathbf{then}\ \uparrow(\mathbf{length}(sndq @ str') - 1)$
$\quad \mathbf{else}\ sndurp)$

Description

From thread *tid*, which is in the Run state, a $\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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 MSG_PEEK and MSG_WAITALL flags are not set in $opts_0$. *space* is the space in the socket's send queue, calculated using send_queue_space (p93).

This rule covers two cases: (1) there is space in the socket's send queue for all the data; and (2) there is not space for all the data but the call is non-blocking (the MSG_DONTWAIT flag is set in *opts* or the socket's O_NONBLOCK flag is set), and the space is greater than zero, or, on FreeBSD, greater than the minimum number of bytes for send() operations on the socket, $sf.n(\text{SO\_SNDLOWAT})$.

In (1) all of the data *str* is appended to the socket's send queue and the returned string, str'', is the empty string. In (2), the first *space* bytes of data, str', are appended to the socket's send queue and the remaining data, str'', is returned.

In both cases a $tid\cdot\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ transition is made, leaving the thread state $\textsc{Ret}(\text{OK}(\mathbf{implode}\ str''))$. If the data was marked as out-of-band, $\text{MSG\_OOB} \in opts$, then the socket's send urgent pointer will point to the end of the send queue.

Model details

The data to be sent is of type string in the 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.

send_2 **tcp: block Block waiting for space in socket's send queue**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\ socks := socks \oplus [(sid, \mathrm{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore, \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{SEND2}(sid, *, str, opts))_{\mathrm{never\_timer}});\ socks := socks \oplus [(sid, \mathrm{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore, \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle$$

$opts = \mathbf{list\_to\_set}\ opts_0\ \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff) \wedge$
$\neg((\neg\ \mathrm{bsd\_arch}\ h.arch \wedge \mathrm{MSG\_DONTWAIT} \in opts) \vee ff.b(\mathrm{O\_NONBLOCK})) \wedge$

$space \in \mathrm{send\_queue\_space}$
$\quad (sf.n(\mathrm{SO\_SNDBUF}))(\mathbf{length}\ sndq)(\mathrm{MSG\_OOB} \in opts)h.arch\ cb.t\_maxseg\ i_2 \wedge$

$(\{\mathrm{MSG\_PEEK}; \mathrm{MSG\_WAITALL}\} \cap opts = \emptyset \vee \mathrm{bsd\_arch}\ h.arch) \wedge$

$((st \in \{\mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}\} \wedge$
$\quad space < \mathbf{length}\ str) \vee$
$(\mathrm{linux\_arch}\ h.arch \wedge st \in \{\mathrm{SYN\_SENT}; \mathrm{SYN\_RECEIVED}\}))$

Description

From thread tid, which is in the RUN state, a $\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ call is made. fd refers to a TCP socket sid that has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The call is a blocking one: the socket's O_NONBLOCK flag is not set and the MSG_DONTWAIT flag is not set in $opts_0$. The MSG_PEEK and MSG_WAITALL flags are not set in $opts_0$.

The space in the socket's send queue, *space* (calculated using send_queue_space (p93)), is less than the length in bytes of the data to be sent, *str*.

The call blocks, leaving the thread state $\mathrm{SEND2}(sid, *, str, opts)$ via a $tid \cdot \mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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**

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{\tau}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\mathbf{implode}\ str''')))_{\text{sched\_timer}});\\
\quad socks := socks \oplus\\
\qquad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *, sndq \mathbin{@} str', sndurp', rcvq, rcvurp, iobc)))]|\!\rangle
\end{array}
$$

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

Description

Thread *tid* is blocked in state $\text{SEND2}(sid, *, str, opts)$ where the TCP socket *sid* has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT.

The space in the socket's send queue, *space* (calculated using send_queue_space (p93)), is greater than or equal to the length of the data to be sent, *str*. The data is appended to the socket's send queue and the call successfully returns the empty string. A τ transition is made, leaving the thread state RET(OK“”). If the data was marked as out-of-band, $\text{MSG\_OOB} \in opts$, then the socket's urgent pointer will be updated to point to the end of the socket's send queue.

Model details

The data to be sent is of type string in the send() call but is a *byte* list when the datagram is constructed. Here the data, *str* is of type *byte* list and in the transition **implode** *str* is used to convert it into a string.

send_3a **tcp: block** **From blocked state, transfer some data to the send queue and remain blocked**

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{\tau}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str'', opts))_{\text{never\_timer}});\\
\quad socks := socks \oplus\\
\qquad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,\\
\qquad\qquad \text{TCP\_Sock}(st, cb, *, sndq \mathbin{@} str', sndurp', rcvq, rcvurp, iobc)))]|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
st \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \wedge\\
space \in \text{send\_queue\_space}\\
\qquad (sf.n(\text{SO\_SNDBUF}))(\mathbf{length}\ sndq)(\text{MSG\_OOB} \in opts)h.arch\ cb.t\_maxseg\ i_2 \wedge\\
space < \mathbf{length}\ str \wedge space > 0 \wedge\\
(str', str'') = \text{SPLIT}\ space\ str \wedge\\
sndurp' = \mathbf{if}\ \text{MSG\_OOB} \in opts\ \mathbf{then}\ \uparrow(\mathbf{length}(sndq \mathbin{@} str') - 1)\ \mathbf{else}\ sndurp
\end{array}
$$

Description

Thread *tid* is blocked in state $\text{SEND2}(sid, *, str, opts)$ where TCP socket *sid* has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT. The amount of space in the socket's send queue, *space* (calculated using send_queue_space (p93)), is less than the length of the remaining data to be sent, *str*, and greater than 0. The socket's send queue is filled by appending the first *space* bytes of *str*, str', to it.

A τ transition is made, leaving the thread state $\text{SEND2}(sid, *, str'', opts)$ where str'' is the remaining data to be sent. If the data in *str* is out-of-band, MSG_OOB is set in *opts*, then the socket's urgent pointer is updated to point to the end of the socket's send queue.

Note it is unclear whether or not MSG_OOB should be removed from *opts* in the state.

send_4 **tcp: fast fail** **Fail with EAGAIN: non-blocking semantics requested and call would block**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid \cdot \mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched\_timer}})\rangle\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$h.socks[sid] = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$

$(\{\text{MSG\_PEEK}; \text{MSG\_WAITALL}\} \cap opts = \emptyset \vee \text{bsd\_arch}\ h.arch) \wedge$

$((\neg\, \text{bsd\_arch}\ h.arch \wedge \text{MSG\_DONTWAIT} \in opts) \vee ff.b(\text{O\_NONBLOCK})) \wedge$

$((st \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \wedge$
$\quad space \in \text{send\_queue\_space}$
$\qquad (sf.n(\text{SO\_SNDBUF}))(\mathbf{length}\ sndq)(\text{MSG\_OOB} \in opts)h.arch\ cb.t\_maxseg\ i_2 \wedge$
$\quad \neg(space \geq \mathbf{length}\ str \vee (\mathbf{if}\ \text{bsd\_arch}\ h.arch\ \mathbf{then}\ space \geq sf.n(\text{SO\_SNDLOWAT})\ \mathbf{else}\ space > 0))) \vee$
$(st \in \{\text{SYN\_SENT}; \text{SYN\_RECEIVED}\} \wedge$
$\quad \text{linux\_arch}\ h.arch))$

Description

From thread *tid*, which is in the RUN state, a $\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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 (p93)), is less than both the length of the data to send *str*; and on FreeBSD is less than the minimum number of bytes for socket send operations, $sf.n(\text{SO\_SNDLOWAT})$, or on Linux and WinXP is equal to zero. The call would have to block, but because it is non-blocking, it fails with an EAGAIN error.

A $tid \cdot \mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state RET(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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Run})_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle \xrightarrow{tid\cdot\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Ret}(\text{FAIL}\ e))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle$$

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

Description

From thread tid, which is in the Run state, a $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid\cdot\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts)$ transition is made, leaving the thread in state Ret(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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Send2}(sid, *, str, opts))_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle \xrightarrow{\tau} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Ret}(\text{FAIL}\ e))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle$$

$\text{proto\_of}\ sock.pr = \text{PROTO\_TCP}$

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(FAIL e).

send_6 **tcp: fast fail** **Fail with ENOTCONN or EPIPE: socket not connected**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Run})_d)|\!\rangle \xrightarrow{tid\cdot\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{Ret}(\text{FAIL}\ err))_{\text{sched\_timer}})|\!\rangle$$

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

$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\textsc{FT\_Socket}(sid), ff) \wedge$
$sock = (h.socks[sid]) \wedge$
$\text{TCP\_PROTO}(tcp\_sock) = sock.pr \wedge$
$sock.es = * \wedge$
$(tcp\_sock.st \in \{\text{CLOSED}; \text{LISTEN}\} \vee$

$(tcp\_sock.st \in \{\text{SYN\_SENT}; \text{SYN\_RECEIVED}\} \wedge \neg(\text{linux\_arch } h.arch)) \vee$
F (* Placeholder for: if `tcp_disconnect` or `tcp_usrclose` has been invoked *)
$) \wedge$
$err = (\textbf{if } \text{linux\_arch } h.arch \textbf{ then } \text{EPIPE} \textbf{ else } \text{ENOTCONN})$

Description

From thread *tid*, which is in the RUN state, a $\mathsf{send}(fd, *, \textbf{implode } str, opts_0)$ 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 $tid\cdot\mathsf{send}(fd, *, \textbf{implode } str, opts_0)$ 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**

$$
\begin{array}{l}
h \, \langle\!| ts := ts \oplus (tid \mapsto (t)_d); \\
\quad socks := socks \oplus \\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{TCP\_PROTO}(tcp)))] |\!\rangle \\
\xrightarrow{lbl} h \, \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } err))_{\text{sched\_timer}}); \\
\quad socks := socks \oplus \\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{TCP\_PROTO}(tcp)))] |\!\rangle
\end{array}
$$

$$
\left(
\left(
\begin{array}{l}
\exists fd\ ff\ str\ opts_0\ i_2\ p_2. \\
\quad fd \in \mathbf{dom}(h.fds) \wedge \\
\quad fid = h.fds[fd] \wedge \\
\quad h.files[fid] = \textsc{File}(\textsc{FT\_Socket}(sid), ff) \wedge \\
\quad t = \textsc{Run} \wedge \\
\quad lbl = tid\cdot\mathsf{send}(fd, *, \textbf{implode } str, opts_0) \wedge \\
\quad rc = \textsc{fast fail} \wedge \\
\quad is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2 \wedge \\
\quad (\textbf{if } tcp.st \neq \text{CLOSED} \textbf{ then} \\
\qquad \exists i_1\ p_1. is_1 = \uparrow i_1 \wedge ps_1 = \uparrow p_1 \\
\qquad \textbf{else } \mathbf{T})
\end{array}
\right)
\vee
\left(
\begin{array}{l}
\exists opts\ str. \\
\quad t = \textsc{Send2}(sid, *, str, opts) \wedge \\
\quad lbl = \tau \wedge \\
\quad rc = \textsc{slow}\ \text{urgent}\ \textsc{fail}
\end{array}
\right)
\right) \wedge
$$

$(\textbf{if } \text{windows\_arch } h.arch \textbf{ then } err = \text{ESHUTDOWN}$
$\textbf{else } err = \text{EPIPE})$

Description

This rule covers two cases: (1) from thread *tid*, which is in the Run state, a $\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ call is made; and (2) thread *tid* is blocked in state $\text{Send2}(sid, *, str, opts)$. In (1), fd refers to a TCP socket *sid* that has binding quad $(is_1, ps_1, \uparrow i_2, \uparrow p_2)$. In both cases the socket is shutdown for writing. The call fails with an EPIPE error.

The thread is left in state Ret(FAIL EPIPE), via a $tid\cdot\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{Run})_d)\rangle\!\rangle \xrightarrow{tid\cdot\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{Ret}(\text{FAIL EOPNOTSUPP}))_{\text{sched\_timer}})\rangle\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{File}(\text{FT\_Socket}(sid), ff) \wedge$
$\text{proto\_of}(h.socks[sid]).pr = \text{PROTO\_TCP} \wedge$
$opts = \mathbf{list\_to\_set}\ opts_0 \wedge$
$(\text{MSG\_PEEK} \in opts \vee \text{MSG\_WAITALL} \in opts) \wedge$
$\neg\,\text{bsd\_arch}\ h.arch$

Description

From thread *tid*, which is in the Run state, a $\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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 $tid\cdot\mathsf{send}(fd, *, \mathbf{implode}\ str, opts_0)$ 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.

15.22 send() (UDP only)

$\mathsf{send} : (\mathsf{fd} * (\mathsf{ip} * \mathsf{port})\ \mathsf{option} * \mathsf{string} * \mathsf{msgbflag\ list}) \rightarrow \mathsf{string}$

This section describes the behaviour of send() for UDP sockets. A call to $\mathsf{send}(\mathsf{fd}, addr, data, flags)$ 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.

15.22.1 Errors

In addition to errors returned via ICMP (see *deliver_in_icmp_3* (p337)), a call to send() can fail with the errors below, in which case the corresponding exception is raised:

EADDRINUSE	The socket's peer address is not set and the destination address specified would give the socket a binding quad i_1, p_1, i_2, p_2 which is already in use by another socket.
EADDRNOTAVAIL	There are no ephemeral ports left for autobinding to.
EAGAIN	The 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.

15.22.2 Common cases

send_9; *return_1*;

15.22.3 API

Posix:	`ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr socklen_t dest_len);`
FreeBSD:	`ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);`
Linux:	`int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);`
WinXP:	`int sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen);`

In the Posix interface:

- `socket` is the file descriptor of the socket to send from, corresponding to the fd argument of the model send().
- `message` is a pointer to the data to be sent of length `length`. The two together correspond to the string argument of the model send().
- `flags` is an OR of the message flags for the send() call, corresponding to the msgbflag list in the model send().
- `dest_addr` and `dest_len` correspond to the *addr* argument of the model send(). `dest_addr` is either null or a pointer to a sockaddr structure containing the destination address for the data. If it is null it corresponds to $addr = *$. If it contains an address, then it corresponds to $addr = \uparrow(i_3, p_3)$ 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()`.

15.22.4 Model details

If the call blocks then the thread enters state $\text{SEND2}(\mathsf{sid}, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts)$ where

- sid : sid is the identifier of the socket that made the send() call,
- $addr$: $(\mathsf{ip} * \mathsf{port})$ 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.

15.22.5 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

15.22.6 Rules

send_9 **udp: fast succeed** **Enqueue datagram and return successfully**

$$h_0 \xrightarrow{tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{OK}(``")))_{\mathrm{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!\langle es := es; ps_1 := \uparrow p_1'; pr := \mathrm{UDP\_PROTO}(udp)\rangle\!\rangle)];\ bound := bound;\ oq := oq' \rangle\!\rangle$$

$$\begin{aligned}
&h_0 = h\ \langle\!\langle\ ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\\
&\qquad socks := socks \oplus\\
&\qquad\quad [(sid, sock\ \langle\!\langle\ es := es; pr := \mathrm{UDP_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle \land\\
&fd\ \in \mathbf{dom}(h_0.fds) \land\\
&fid = h_0.fds[fd] \land\\
&h_0.files[fid] = \mathrm{FILE}(\mathrm{FT_SOCKET}(sid), ff) \land\\
&sock.cantsndmore = \mathbf{F} \land\\
&STRLEN(\mathbf{implode}\ str) \leq \mathrm{UDPpayloadMax}\ h_0.arch \land\\
&((addr \neq *) \lor (sock.is_2 \neq *)) \land\\
&p_1'\ \in\ \mathrm{autobind}(sock.ps_1, \mathrm{PROTO_UDP}, h_0.socks) \land\\
&(\mathbf{if}\ sock.ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \land\\
&\mathrm{dosend}(h.ifds, h.rttab, (addr, str), (sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{T}) \land\\
&(\mathbf{if}\ \mathrm{bsd_arch}\ h.arch\ \mathbf{then}\ (h_0.socks[sid]).sf.n(\mathrm{SO_SNDBUF}) \geq STRLEN(\mathbf{implode}\ str)\\
&\quad \mathbf{else}\ \mathrm{MSG_OOB} \notin (\mathbf{list_to_set}\ opts_0)) \land\\
&(\neg(\mathrm{windows_arch}\ h.arch) \implies es = *)
\end{aligned}$$

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 $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ succeeds if:

- the length of *str* is less than UDPpayloadMax (p70), the architecture-dependent maximum payload for a UDP datagram.
- The socket has a peer IP address set in its is_2 field or the *addr* argument is $\uparrow(i_3, p_3)$, specifying a destination address.
- The socket is bound to a local port p_1', or it can be autobound to p_1' and *sid* added to the list of bound sockets.
- A UDP datagram is constructed from the socket's binding quad $(sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2)$, the destination address argument *addr*, and the data *str*. This datagram is successfully enqueued on the outqueue of the host, *oq* to form outqueue oq' using auxiliary function dosend (p96).

A $tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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**

$$h_0$$

$$\xrightarrow{tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)}$$

$$\begin{array}{l} h\ \langle\!|ts := \\ ts \oplus (tid \mapsto \text{TIMED}(\text{SEND2}(sid, \uparrow(addr, sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), \\ \qquad\qquad str, opts), \\ \qquad\qquad \text{never\_timer})); \\ socks := socks \oplus \\ \quad [(sid, sock\ \langle\!|es := es; ps_1 := \uparrow p_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]; \\ bound := bound; \\ oq := oq'|\!\rangle \end{array}$$

$$\begin{array}{l} h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ \qquad socks := socks \oplus \\ \qquad\quad [(sid, sock\ \langle\!|\ es := es; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle \wedge \\ fd\ \in \mathbf{dom}(h_0.fds) \wedge \\ fid = h_0.fds[fd] \wedge \\ h_0.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge \\ sock.cantsndmore = \mathbf{F} \wedge \\ (\neg(\text{windows\_arch}\ h.arch) \implies es = *) \wedge \\ opts = \mathbf{list\_to\_set}\ opts_0 \wedge \\ \neg((\neg\,\text{bsd\_arch}\ h.arch \wedge \text{MSG\_DONTWAIT} \in opts) \vee ff.b(\text{O\_NONBLOCK})) \wedge \\ ((\text{linux\_arch}\ h.arch \vee \text{windows\_arch}\ h.arch) \implies \text{MSG\_OOB} \notin opts) \wedge \\ p_1' \in \text{autobind}(sock.ps_1, \text{PROTO\_UDP}, h_0.socks) \wedge \\ (\mathbf{if}\ sock.ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \wedge \\ \text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F}) \wedge \\ ((addr \neq *) \vee (sock.is_2 \neq *)) \end{array}$$

Description

Consider a UDP socket *sid* referenced by *fd* that is not shutdown for writing and has no pending errors. A $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $\text{SEND2}(sid, \uparrow(addr, sock.is_1, \uparrow p_1', sock.is_2, sock.ps_2), str, opts)$. 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**

$$h_0 \xrightarrow{tid \cdot \mathsf{send}(fd, addr, \textbf{implode}\ str, opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{sched\_timer}}); socks := socks \oplus [(sid, sock\ \langle\!|es := es; ps_1 := \uparrow p_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]; bound := bound; oq := oq'|\!\rangle$$

$$\begin{aligned}
&h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
&\quad socks := socks \oplus \\
&\quad\quad [(sid, sock\ \langle\!|\ es := es; pr := \text{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle\ \wedge \\
&fd\ \in \mathbf{dom}(h_0.fds) \wedge \\
&fid = h_0.fds[fd] \wedge \\
&h_0.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
&sock.cantsndmore = \mathbf{F} \wedge \\
&(\neg(\text{windows_arch}\ h.arch) \implies es = *) \wedge \\
&p_1' \in \text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks) \wedge \\
&(\textbf{if}\ sock.ps_1 = *\ \textbf{then}\ bound = sid :: h_0.bound\ \textbf{else}\ bound = h_0.bound) \wedge \\
&((addr \neq *) \vee (sock.is_2 \neq *)) \wedge \\
&opts = \textbf{list_to_set}\ opts_0 \wedge \\
&((\neg\,\text{bsd_arch}\ h.arch \wedge \text{MSG_DONTWAIT} \in opts) \vee ff.b(\text{O_NONBLOCK})) \wedge \\
&\text{dosend}(h_0.ifds, h_0.rttab, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h_0.oq, oq', \mathbf{F})
\end{aligned}$$

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 $\mathsf{send}(fd, addr, \textbf{implode}\, str, opts_0$ 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 $tid \cdot \mathsf{send}(fd, addr, \textbf{implode}\ str, opts_0)$ 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**

$$h_0$$

$$\xrightarrow{tid \cdot \mathsf{send}(fd, *, \textbf{implode}\ str, opts_0)}$$

$$h\ \langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ err))_{\text{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1', *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))];$$
$$bound := bound |\!\rangle$$

$$h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\text{RUN})_d);$$
$$socks := socks \oplus$$
$$[(sid, \text{SOCK}(\uparrow fid, sf, is_1, ps_1, *, *, es, cantsndmore, cantrcvmore, \text{UDP\_PROTO}(udp)))] |\!\rangle \wedge$$
$$fd \in \textbf{dom}(h.fds) \wedge$$
$$fid = h.fds[fd] \wedge$$
$$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$$
$$(\textbf{if}\ \text{bsd\_arch}\ h.arch\ \textbf{then}\ err = \text{EDESTADDRREQ}$$
$$\textbf{else}\ err = \text{ENOTCONN}) \wedge$$
$$(\neg(\text{windows\_arch}\ h.arch) \implies es = *) \wedge$$
$$(\textbf{if}\ \text{linux\_arch}\ h.arch\ \textbf{then}$$
$$\exists p_1'.p_1' \in \text{autobind}(ps_1, \text{PROTO\_UDP}, h_0.socks) \wedge ps_1' = \uparrow p_1' \wedge$$
$$(\textbf{if}\ ps_1 = *\ \textbf{then}\ bound = sid :: h_0.bound\ \textbf{else}\ bound = h_0.bound)$$
$$\textbf{else}\ bound = h_0.bound \wedge ps_1' = ps_1)$$

Description

Consider a UDP socket *sid* referenced by *fd* that has no pending errors.

A call $\mathsf{send}(fd, addr, \textbf{implode}\ str, opts_0$ 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 $tid \cdot \mathsf{send}(fd, *, \textbf{implode}\ str, opts_0)$ 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

$$h_0 \xrightarrow{tid \cdot \mathsf{send}(fd,\, addr,\, \mathbf{implode}\ str,\, opts_0)} h\ \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL\ EMSGSIZE}))_{\mathrm{sched\_timer}}); socks := socks \oplus [(sid, sock\ \langle\!| ps_1 := ps_1'; pr := \mathrm{UDP\_PROTO}(udp) |\!\rangle)]; bound := bound |\!\rangle$$

$$\begin{aligned}
&h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\\
&\qquad socks := socks \oplus\\
&\qquad\quad [(sid, sock\ \langle\!|\ pr := \mathrm{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle \wedge\\
&fd\ \in \mathbf{dom}(h_0.fds) \wedge\\
&fid = h_0.fds[fd] \wedge\\
&h_0.files[fid] = \mathrm{FILE}(\mathrm{FT_SOCKET}(sid), ff) \wedge\\
&(STRLEN(\mathbf{implode}\ str) > \mathrm{UDPpayloadMax}\ h_0.arch \vee\\
&\quad (\mathrm{bsd_arch}\ h.arch \wedge STRLEN(\mathbf{implode}\ str) > (h_0.socks[sid]).sf.n(\mathrm{SO_SNDBUF}))) \wedge\\
&ps_1'\ \in \{sock.ps_1\} \cup (\mathbf{image}(\uparrow)(\mathrm{autobind}(sock.ps_1, \mathrm{PROTO_UDP}, h_0.socks))) \wedge\\
&(\mathbf{if}\ sock.ps_1 = * \wedge ps_1' \neq *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound)
\end{aligned}$$

Description

Consider a UDP socket *sid* referenced by *fd*. A call $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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**

$$\begin{aligned}
&h\ \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\\
&socks := socks \oplus\\
&\quad [(sid, \mathrm{SOCK}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \mathrm{UDP_PROTO}(udp)))]|\!\rangle\\
&\xrightarrow{tid \cdot \mathsf{send}(fd,\, addr,\, \mathbf{implode}\ str,\, opts_0)}\\
&h\ \langle\!| ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{sched_timer}});\\
&socks := socks \oplus\\
&\quad [(sid, \mathrm{SOCK}(\uparrow fid, sf, *, *, *, *, es, cantsndmore, cantrcvmore, \mathrm{UDP_PROTO}(udp)))]|\!\rangle
\end{aligned}$$

$$\begin{aligned}
&fd\ \in \mathbf{dom}(h.fds) \wedge\\
&fid = h.fds[fd] \wedge\\
&h.files[fid] = \mathrm{FILE}(\mathrm{FT_SOCKET}(sid), ff) \wedge\\
&cantsndmore = \mathbf{F} \wedge
\end{aligned}$$

$(\neg(\text{windows\_arch}\ h.arch) \implies es = *) \wedge$
$\text{autobind}(*, \text{PROTO\_UDP}, h.socks) = \emptyset \wedge$
$e \in \{\text{EAGAIN}; \text{EADDRNOTAVAIL}; \text{ENOBUFS}\}$

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 $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid{\cdot}\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d);$$
$$socks := socks \oplus$$
$$[(sid, sock\ \langle\!\langle es := es; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle$$
$$\xrightarrow{\tau}\ h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(``")))_{\text{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, sock\ \langle\!\langle es := es; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)];$$
$$oq := oq'\rangle\!\rangle$$

$sock.cantsndmore = \mathbf{F} \wedge$
$(\neg(\text{windows\_arch}\ h.arch) \implies es = *) \wedge$
$STRLEN(\mathbf{implode}\ str) \leq \text{UDPpayloadMax}\ h.arch \wedge$
$(\text{dosend}(h.ifds, h.rttab, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T}) \vee$
$\quad \text{dosend}(h.ifds, h.rttab, (addr, str), (sock.is_1, sock.ps_1, sock.is_2, sock.ps_2), h.oq, oq', \mathbf{T})) \wedge$
$(addr \neq * \vee sock.is_2 \neq * \vee is_2 \neq *)$

Description

Consider a UDP socket *sid* that is not shutdown for writing and has no pending errors. The thread *tid* is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

A datagram can be constructed using *str* as its data. The length in bytes of *str* is less than or equal to UDPpayloadMax, the architecture-dependent maximum payload size for a UDP datagram. There are three possible destination addresses:

- *addr*, the destination address specified in the 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 $\text{RET}(\text{OK}(``"))$, and the host with new outqueue oq'.

send_16 **udp: slow urgent fail** **Fail: blocked socket has entered an error state**

$$h \ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str))_d); \\ socks := socks \oplus \\ \quad [(sid, sock \ \langle\!\langle es := \uparrow e; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle$$

$$\xrightarrow{\tau} \ h \ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched\_timer}}); \\ socks := socks \oplus \\ [(sid, sock \ \langle\!\langle es := *; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle$$

$$\neg(\text{windows\_arch } h.arch)$$

Description

Consider a UDP socket sid that has pending error $\uparrow e$. The thread tid is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$. The error, e, will be returned to the caller.

At τ transition is made, leaving the thread state $RET(\text{FAIL } e)$.

Note that the error has occurred after the thread entered the SEND2 state: rule *send_11* specifies that the call cannot block if there is a pending error.

Variations

WinXP	This rule does not apply: all pending errors on a socket are ignored for a 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**

$$h \ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str, opts))_d); \\ socks := socks \oplus \\ \quad [(sid, sock \ \langle\!\langle sf := sf; es := es; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle$$

$$\xrightarrow{\tau} \ h \ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL } e))_{\text{sched\_timer}}); \\ socks := socks \oplus \\ \quad [(sid, sock \ \langle\!\langle sf := sf; es := es; pr := \text{UDP\_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle$$

$$(\neg(\text{windows\_arch } h.arch) \implies es = *) \land \\ (\exists oq'. \text{dosend}(h.ifds, h.rttab, (addr, str), (is_1, ps_1, is_2, ps_2), h.oq, oq', \mathbf{T})) \land \\ ((STRLEN(\textbf{implode } str) > \text{UDPpayloadMax } h.arch \land (e = \text{EMSGSIZE})) \lor \\ \quad (\text{bsd\_arch } h.arch \land STRLEN(\textbf{implode } str) > sf.n(\text{SO\_SNDBUF}) \land (e = \text{EMSGSIZE})) \lor \\ \quad ((sock.is_2 = *) \land (addr = *) \land (e = \text{ENOTCONN})))$$

Description

Consider a UDP socket sid with no pending errors. The thread tid is blocked in state $\text{SEND2}(sid, \uparrow(addr, is_1, ps_1, is_2, ps_2), str)$.

A datagram is constructed with str as its payload. Its destination address is taken from $addr$, the destination address specified when the send() call was made, or (is_2, ps_2), the socket's peer address when the send() call was made. It is possible to enqueue the datagram on the host's outqueue, $h.oq$.

This rule covers two cases. In the first, the length in bytes of str is greater than UDPpayloadMax, the architecture-dependent maximum payload size for a UDP datagram. The error EMSGSIZE is returned.

In the second case, the original send() call did not have a destination address specified: $addr = *$, and the socket has had the IP address of its peer address unset: $sock.is_2 = *$. The peer address of the socket when the send() call was made, (is_2, ps_2), is ignored, and an ENOTCONN error is returned.

In either case, a τ transition is made, leaving the thread state $\textsc{Ret}(\text{FAIL } e)$ where e is either EMSGSIZE or ENOTCONN.

Variations

FreeBSD	An EMSGSIZE error can also be returned if the size of *str* is greater than the value of the socket's SO_SNDBUF option.
WinXP	Any pending error on the socket is ignored.

send_18 **udp: fast fail** **Fail with EOPNOTSUPP: MSG_PEEK flag not supported for send() calls on WinXP; or MSG_OOB flag not supported on WinXP and Linux**

$$h_0 \xrightarrow{tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EOPNOTSUPP}))_{\text{sched\_timer}}); socks := socks \oplus [(sid, sock\ \langle\!| ps_1 := ps_1'; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]; bound := bound|\!\rangle$$

$$\begin{array}{l}
h_0 = h\ \langle\!|\ ts := ts \oplus (tid \mapsto (\textsc{Run})_d); \\
\quad socks := socks \oplus \\
\quad [(sid, sock\ \langle\!|\ ps_1 := ps_1; pr := \text{UDP_PROTO}(udp)|\!\rangle)]|\!\rangle \wedge \\
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \textsc{File}(\text{FT_}\textsc{Socket}(sid), ff) \wedge \\
opts = \mathbf{list_to_set}\ opts_0 \wedge \\
((\text{MSG_PEEK} \in opts \wedge \text{windows_arch}\ h.arch) \vee \\
(\text{MSG_OOB} \in opts \wedge sock.cantsndmore = \mathbf{F} \wedge (\text{linux_arch}\ h.arch \vee \text{windows_arch}\ h.arch))) \wedge \\
(\mathbf{if}\ \text{linux_arch}\ h.arch\ \mathbf{then} \\
\quad \exists p_1'.p_1' \in \text{autobind}(ps_1, \text{PROTO_UDP}, h_0.socks) \wedge ps_1' = \uparrow p_1' \wedge \\
\quad (\mathbf{if}\ ps_1 = *\ \mathbf{then}\ bound = sid :: h_0.bound\ \mathbf{else}\ bound = h_0.bound) \\
\mathbf{else} \\
\quad ps_1 = ps_1' \wedge bound = h_0.bound)
\end{array}$$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the $\textsc{Run}$ state, a $\mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ 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 $tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\ str, opts_0)$ transition is made, leaving the thread in state $\textsc{Ret}(\text{FAIL EOPNOTSUPP})$.

Model details

The $opts_0$ argument is of type list. In the model it is converted to a set *opts* using **list_to_set**. 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().

send_19 **udp: fast fail** **Fail with EADDRINUSE: on FreeBSD, local and destination address quad in use by another socket**

$$h_0 \xrightarrow{tid \cdot \mathsf{send}(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EADDRINUSE}))_{\text{sched\_timer}}); socks := socks \oplus [(sid, sock)]; bound := bound \rangle\!\rangle$$

$$\begin{array}{l}
\text{bsd_arch}\ h.arch \wedge \\
h_0 = h\ \langle\!\langle\ ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
\qquad socks := socks \oplus \\
\qquad\quad [(sid, sock)]\rangle\!\rangle \wedge \\
sock.cantsndmore = \mathbf{F} \wedge \\
(\neg(\text{windows_arch}\ h.arch) \implies sock.es = *) \wedge \\
p_1' \in \text{autobind}(sock.ps_1, \text{PROTO_UDP}, h_0.socks) \wedge \\
(\textbf{if}\ sock.ps_1 = *\ \textbf{then}\ bound = sid :: h_0.bound\ \textbf{else}\ bound = h_0.bound) \wedge \\
i_1' \in \text{auto_outroute}(i_2, sock.is_1, h_0.rttab, h_0.ifds) \wedge \\
fd \in \textbf{dom}(h_0.fds) \wedge \\
fid = h_0.fds[fd] \wedge \\
h_0.files[fid] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge \\
sock = (h_0.socks[sid]) \wedge \\
\text{proto_of}\ sock.pr = \text{PROTO_UDP} \wedge \\
(\exists sid'. \\
\quad sid' \in \textbf{dom}(h_0.socks) \wedge \\
\quad \textbf{let}\ s = h_0.socks[sid']\ \textbf{in} \\
\quad s.is_1 = \uparrow i_1' \wedge s.ps_1 = \uparrow p_1' \wedge \\
\quad s.is_2 = \uparrow i_2 \wedge s.ps_2 = \uparrow p_2 \wedge \\
\quad \text{proto_of}\ s.pr = \text{PROTO_UDP})
\end{array}$$

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 $\mathsf{send}(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)$ 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 $tid \cdot \mathsf{send}(fd, \uparrow(i_2, p_2), \textbf{implode}\ str, opts_0)$ transition will be made, leaving the thread state RET(FAIL EADDRINUSE).

Variations

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

send_21 **udp: fast fail** **Fail with EISCONN: socket has peer address set and destination address is specified in call on FreeBSD**

$$\begin{array}{l}
h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\
socks := socks \oplus \\
\quad [(sid, sock\ \langle\!\langle es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP_PROTO}(udp)\rangle\!\rangle)]\rangle\!\rangle
\end{array}$$

$$\xrightarrow{tid \cdot \mathsf{send}(fd, \uparrow(i_3, p_3), \textbf{implode}\ str, opts_0)}$$

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EISCONN}))_{\text{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, sock \; \langle\!| es := *; is_2 := \uparrow i_2; ps_2 := \uparrow p_2; pr := \text{UDP\_PROTO}(udp) |\!\rangle)] |\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
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 $\mathsf{send}(fd, \uparrow(i_3, p_3), \mathbf{implode}\; str, opts_0)$ 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 $tid \cdot \mathsf{send}(fd, \uparrow(i_3, p_3), \mathbf{implode}\; str, opts_0)$ transition will be made, leaving the thread state Ret(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**

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Run})_d);$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{UDP\_PROTO}(udp)))] |\!\rangle$$

$$\xrightarrow{tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\; str, opts_0)}$$

$$h \; \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } err))_{\text{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, \mathbf{T}, cantrcvmore, \text{UDP\_PROTO}(udp)))] |\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
if windows_arch $h.arch$ **then** $err = \text{ESHUTDOWN}$
else $err = \text{EPIPE}$

Description

From thread *tid*, which is in the Run state, a $\mathsf{send}(fd, addr, \mathbf{implode}\; str, opts_0)$ 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 $tid \cdot \mathsf{send}(fd, addr, \mathbf{implode}\; str, opts_0)$ transition is made, leaving the thread in state Ret(FAIL EPIPE).

Variations

WinXP	The call fails with an ESHUTDOWN error rather than EPIPE.

send_23 **udp: fast fail** **Fail with pending error**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\ socks := socks \oplus [(sid, sock\ \langle\!|es := \uparrow e|\!\rangle)]|\!\rangle \xrightarrow{tid\cdot\mathsf{send}(fd,\ addr,\ \mathbf{implode}\ \ str,\ opts_0)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{sched\_timer}});\ socks := socks \oplus [(sid, sock\ \langle\!|es := *|\!\rangle)]|\!\rangle$$

$fd \in \mathbf{dom}(h.fds)\ \wedge$
$fid = h.fds[fd]\ \wedge$
$h.files[fid] = \mathrm{FILE}(\mathrm{FT\_SOCKET}(sid), ff)\ \wedge$
$\mathrm{proto\_of}\ \ sock.pr = \mathrm{PROTO\_UDP}\ \wedge$
$\neg(\mathrm{windows\_arch}\ \ h.arch)$

Description

From thread *tid*, which is in the RUN state, a $\mathsf{send}(fd, addr, \mathbf{implode}\ \ str, opts_0)$ 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 $tid\cdot\mathsf{send}(fd, addr, \mathbf{implode}\ \ str, opts_0)$ transition is made, leaving the thread in state RET(FAIL e).

Variations

WinXP	This rule does not apply: all pending errors are ignored for send() calls on WinXP.

15.23 setfileflags() (TCP and UDP)

setfileflags : (fd ∗ filebflag list) → unit

A call to setfileflags(fd, *flags*) sets the flags on a file referred to by fd. *flags* is the list of file flags to set. The possible flags are:

- O_ASYNC Specifies whether signal driven I/O is enabled.
- O_NONBLOCK Specifies whether a socket is non-blocking.

The call returns successfully if the flags were set, or fails with an error otherwise.

15.23.1 Errors

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

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

15.23.2 Common cases

setfileflags_1; *return_1*

15.23.3 API

setfileflags() is Posix `fcntl(fd,F_GETFL,flags)`. On WinXP it is `ioctlsocket()` with the `FIONBIO` command.

Posix:	`int fcntl(int fildes, int cmd, ...);`
FreeBSD:	`int fcntl(int fd, int cmd, ...);`
Linux:	`int fcntl(int fd, int cmd);`
WinXP:	`int ioctlsocket(SOCKET s, long cmd, u_long* argp)`

In the Posix interface:

- `fildes` is a file descriptor for the file to retrieve flags from. It corresponds to the `fd` argument of the model setfileflags(). On WinXP the `s` is a socket descriptor corresponding to the `fd` argument of the model setfileflags().
- `cmd` is a command to perform an operation on the file. This is set to `F_GETFL` for the model setfileflags(). On WinXP, `cmd` is set to `FIONBIO` to get the O_NONBLOCK flag; there is no O_ASYNC flag on WinXP.
- The call takes a variable number of arguments. For the model setfileflags() it takes three arguments: the two described above and a third of type `long` which represents the list of flags to set, corresponding to the *flags* argument of the model setfileflags(). On WinXP this is the `argp` argument.
- The returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

15.23.4 Model details

The following errors are not modelled:

- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.
- `WSAENOTSOCK` is a possible error on WinXP as the `ioctlsocket()` call is specific to a socket. In the model the setfileflags() call is performed on a file.

15.23.5 Summary

setfileflags_1	**all: fast succeed**	Update all the file flags for an open file description

15.23.6 Rules

setfileflags_1 **all: fast succeed** **Update all the file flags for an open file description**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\ files := files \oplus [(fid, \textsc{File}(ft, ff\ \langle\!|b := ffb|\!\rangle))]|\!\rangle$$

$$\xrightarrow{tid\cdot \mathsf{setfileflags}(fd, flags)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\ files := files \oplus [(fid, \textsc{File}(ft, ff\ \langle\!|b := ffb'|\!\rangle))]|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$ffb' = \lambda x.x \in flags$

Rule version: $Id: TCP1_hostLTSScript.sml,v 1.961 2005/03/18 10:34:36 kw217 Exp $

Description

From thread *tid*, which is in the RUN state, a setfileflags($fd, flags$) call is made. *fd* refers to the open file description (fid, FILE(ft, ff ⦇$b := ffb$⦈)) where *ffb* is the set of boolean file flags currently set. *flags* is a list of boolean file flags, possibly containing duplicates.

All of the boolean file flags for the file description will be updated. The flags in *flags* will all be set to **T**, and all other flags will be set to **F**, resulting in a new set of boolean file flags, ffb'.

A *tid*·setfileflags($fd, flags$) transition is made, leaving the thread state RET(OK()).

Note this is not exactly the same as *getfileflags_1*: getfileflags never returns duplicates, but duplicates may be passed to setfileflags.

15.24 setsockbopt() (TCP and UDP)

setsockbopt : (fd ∗ sockbflag ∗ bool) → unit

A call setsockbopt(fd, f, b) sets the value of one of a socket's boolean flags.

Here the fd argument is a file descriptor referring to a socket on which to set a flag, f is the boolean socket flag to set, and b is the value to set it to. Possible boolean flags are:

- SO_BSDCOMPAT Specifies whether the BSD semantics for delivery of ICMPs to UDP sockets with no peer address set is enabled.
- SO_DONTROUTE Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address.
- SO_KEEPALIVE Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol.
- SO_OOBINLINE Leaves received out-of-band data (data marked urgent) inline.
- SO_REUSEADDR Specifies that the rules used in validating addresses supplied to bind() should allow reuse of local ports, if this is supported by the protocol.

15.24.1 Errors

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

ENOPROTOOPT	The option is not supported by the protocol.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.24.2 Common cases

setsockbopt_1; *return_1*

15.24.3 API

setsockbopt() is Posix `setsockopt()` for boolean-valued socket flags.

Posix: `int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);`
FreeBSD: `int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`
Linux: `int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`
WinXP: `int setsockopt(SOCKET s, int level, int optname, const char* optval,int optlen);`

In the Posix interface:

- `socket` is the file descriptor of the socket to set the option on, corresponding to the fd argument of the model 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()`.

15.24.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to 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.

15.24.5 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

15.24.6 Rules

setsockbopt_1 **all: fast succeed** **Successfully set a boolean socket flag**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d);\ socks := socks \oplus [(sid, sock)] \rangle\!\rangle \xrightarrow{tid \cdot \mathsf{setsockbopt}(fd, f, b)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\ socks := socks \oplus [(sid, sock')] \rangle\!\rangle$$

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

$sock' = sock \; ⦇ \; sf := sock.sf \; ⦇ \; b := sock.sf.b \oplus (f \mapsto b) ⦈ ⦈$
$\quad \wedge$

$(\text{windows\_arch} \;\; h.arch \wedge \text{proto\_of} \;\; sock.pr = \text{PROTO\_UDP}$
$\qquad \Longrightarrow \; f \; \notin \{\text{SO\_KEEPALIVE}; \text{SO\_OOBINLINE}\})$

Description

Consider a socket *sid*, referenced by *fd*, and with socket flags *sock.sf*. From thread *tid*, which is in the Run state, a 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**

$$h \; ⦇ ts := ts \oplus (tid \mapsto (\text{Run})_d);$$
$$socks := socks \oplus$$
$$[(sid, sock \; ⦇ pr := \text{UDP\_PROTO}(udp) ⦈)] ⦈$$
$$\xrightarrow{tid \cdot \text{setsockbopt}(fd, f, b)} \; h \; ⦇ ts := ts \oplus (tid \mapsto (\text{Ret}(\text{FAIL ENOPROTOOPT}))_{\text{sched\_timer}});$$
$$socks := socks \oplus$$
$$[(sid, sock \; ⦇ pr := \text{UDP\_PROTO}(udp) ⦈)] ⦈$$

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

Description

On WinXP, consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the Run state, a 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.

15.25 setsocknopt() (TCP and UDP)

setsocknopt : (fd $*$ socknflag $*$ int) $\rightarrow$ 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.

15.25.1 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.

15.25.2 Common cases

setsocknopt_1; *return_1*

15.25.3 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()`.

15.25.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.

- 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.

15.25.5 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

15.25.6 Rules

setsocknopt_1 **all: fast succeed** **Successfully set a numeric socket flag**

$$h \;\langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\; socks := socks \oplus [(sid, sock)]|\!\rangle \xrightarrow{tid \cdot \mathsf{setsocknopt}(fd, f, n)} h \;\langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}});\; socks := socks \oplus [(sid, sock')]|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$n' = \mathbf{max}(\text{sf\_min\_n}\ h.arch\ f)(\mathbf{min}(\text{sf\_max\_n}\ h.arch\ f)(\text{clip\_int\_to\_num}\ n)) \wedge$
$ns = (\mathbf{if}\ \text{bsd\_arch}\ h.arch \wedge f = \text{SO\_SNDBUF} \wedge n' < sock.sf.n(\text{SO\_SNDLOWAT})\ \mathbf{then}$
$\quad (sock.sf.n \oplus (f \mapsto n')) \oplus (\text{SO\_SNDLOWAT} \mapsto n')$
$\quad \mathbf{else}\ sock.sf.n \oplus (f \mapsto n')) \wedge$
$sock' = sock \;\langle\!|\ sf := sock.sf\ \langle\!|\ n := ns|\!\rangle|\!\rangle$

Description

Consider the socket sid, referenced by fd, with numeric socket flags $sock.sf.n$. From the thread tid, which is in the RUN state, a 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 **all: fast fail** **Fail with EINVAL: on FreeBSD numeric socket flags cannot be set to zero**

$h \;\langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle$

$$\xrightarrow{tid \cdot \mathsf{setsocknopt}(fd, f, n)} \quad h \; \langle\!\langle ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINVAL}))_{\text{sched\_timer}}) \rangle\!\rangle$$

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(FAIL EINVAL).

Variations

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

setsocknopt_4 **all: fast fail** **Fail with ENOPROTOOPT: SO_SNDLOWAT not settable on Linux**

$$h \; \langle\!\langle ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \rangle\!\rangle$$
$$\xrightarrow{tid \cdot \mathsf{setsocknopt}(fd, f, n)} \quad h \; \langle\!\langle ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL ENOPROTOOPT}))_{\text{sched\_timer}}) \rangle\!\rangle$$

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(FAIL ENOPROTOOPT).

Variations

FreeBSD	This rule does not apply.
WinXP	This rule does not apply. Note the warning from the Win32 docs (at MSDN setsockopt): "If the setsockopt function is called before the bind function, TCP/IP options will not be checked with TCP/IP until the bind occurs. In this case, the setsockopt function call will always succeed, but the bind function call may fail because of an early setsockopt failing." This is currently unimplemented.

15.26 setsocktopt() (TCP and UDP)

setsocktopt : (fd $*$ socktflag $*$ (int $*$ int) option) $\rightarrow$ 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.

15.26.1 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.

15.26.2 Common cases

setsocktopt_1; *return_1*

15.26.3 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 (int $*$ 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()`.

15.26.4 Model details

The following errors are not modelled:

- `EFAULT` signifies the pointer passed as `option_value` was inaccessible. On WinXP, the error `WSAEFAULT` may also signify that the `optlen` parameter was too small. Note this error is not specified by Posix.
- `EINVAL` signifies the `option_name` was invalid at the specified socket `level`. In the model, typing prevents an invalid flag from being specified in a call to 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.

15.26.5 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

15.26.6 Rules

setsocktopt_1 **all: fast succeed** **Successfully set a time-option socket flag**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle \xrightarrow{tid\cdot \mathsf{setsocktopt}(fd, f, t)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\ socks := socks \oplus [(sid, sock')]|\!\rangle$$

$$\begin{array}{l}
fd \in \mathbf{dom}(h.fds) \wedge \\
fid = h.fds[fd] \wedge \\
h.files[fid] = \textsc{File}(\mathrm{FT_}\textsc{Socket}(sid), ff) \wedge \\
\mathrm{tltimeopt_wf}\ \ t \wedge \\
t' = \mathrm{time_of_tltimeopt}\ \ t \wedge \\
t' \geq 0 \wedge \\
(\mathbf{if}\ f \in \{\mathrm{SO_RCVTIMEO}; \mathrm{SO_SNDTIMEO}\} \wedge t' = 0 \\
\ \mathbf{then}\ \ t'' = \infty \\
\ \mathbf{else}\ \ t'' = t') \wedge \\
(\mathbf{if}\ f = \mathrm{SO_LINGER} \wedge t = \uparrow(s, ns)\ \mathbf{then}\ \ ns = 0\ \ \mathbf{else}\ \ \mathbf{T}) \wedge \\
(f \in \{\mathrm{SO_RCVTIMEO}; \mathrm{SO_SNDTIMEO}\} \implies t'' = \infty \vee t'' \leq \mathrm{sndrcv_timeo_t_max}) \wedge \\
sock' = sock\ \langle\!|\ sf := sock.sf\ \langle\!|\ t := sock.sf.t \oplus (f \mapsto t'')|\!\rangle|\!\rangle
\end{array}$$

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 $tid\cdot$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 **all: fast fail** **Fail with ENOPROTOOPT: on WinXP SO_LINGER not settable for a UDP socket**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid \cdot \mathsf{setsocktopt}(fd, f, t)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOPROTOOPT}))_{\text{sched\_timer}})\rangle\!\rangle$$

windows_arch $h.arch \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
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 $\mathsf{setsocktopt}(fd, f, t)$ 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 $tid \cdot \mathsf{setsocktopt}(fd, f, t)$ 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 **all: fast fail** **Fail with EDOM: timeout value too long to fit in socket structure**

$$h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RUN})_d)\rangle\!\rangle \xrightarrow{tid \cdot \mathsf{setsocktopt}(fd, f, t)} h\ \langle\!\langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EDOM}))_{\text{sched\_timer}})\rangle\!\rangle$$

$f \in \{\text{SO\_RCVTIMEO}; \text{SO\_SNDTIMEO}\} \wedge$
tltimeopt_wf $t \wedge$
$t' = $ time_of_tltimeopt $t \wedge$
$(\mathbf{if}\ t' = 0$
$\ \mathbf{then}\ t'' = \infty$
$\ \mathbf{else}\ t'' = t') \wedge$
$\neg(t'' = \infty \vee t'' \leq \text{sndrcv\_timeo\_t\_max})$

Description

From thread *tid*, which is currently in the RUN state, a $\mathsf{setsocktopt}(fd, f, t)$ 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 $tid \cdot \mathsf{setsocktopt}(fd, f, t)$ call is made, leaving the thread state RET(FAIL EDOM).

Model details

The type of t is $(\mathsf{int} * \mathsf{int})\ \mathsf{option}$, but the type of a time-option socket flag is time. The auxiliary function time_of_tltimeopt is used to do the conversion.

15.27 shutdown() (TCP and UDP)

$\mathsf{shutdown} : (\mathsf{fd} * \mathsf{bool} * \mathsf{bool}) \rightarrow \mathsf{unit}$

A call of shutdown(fd, r, w) shuts down either the read-half of a connection, the write-half of a connection, or both. The fd is a file descriptor referring to the socket to shutdown; the r and w indicate whether the socket should be shut down for reading and writing respectively.

For a TCP socket, shutting down the read-half empties the socket's receive queue, but data will still be delivered to it and subsequent recv() calls will return data. Shutting down the write-half of a TCP connection causes the remaining data in the socket's send queue to be sent and then TCP's connection termination to occur.

For Linux and WinXP, a TCP socket may only be shut down if it is in the ESTABLISHED state; on FreeBSD a socket may be shut down in any state.

For a UDP socket, if the socket is shutdown for reading, data may still be read from the socket's receive queue on Linux, but on FreeBSD and WinXP this is not the case. Shutting down the socket for writing causes subsequent send() calls to fail.

15.27.1 Errors

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

ENOTCONN	The socket is not connected and so cannot be shut down.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.
ENOBUFS	Out of resources.

15.27.2 Common cases

A TCP socket is created and connects to a peer; data is transferred between the two; the socket has no more data to send so calls shutdown() to inform the peer of this: *socket_1*; ...; *connect_1*; ...; *shutdown_1*; *return_1*

15.27.3 API

Posix:	`int shutdown(int socket, int how);`
FreeBSD:	`int shutdown(int s, int how);`
Linux:	`int shutdown(int s, int how);`
WinXP:	`int shutdown(SOCKET s, int how);`

In the Posix interface:

- `socket` is a file descriptor referring to the socket to shut down. This corresponds to the fd argument of the model shutdown().
- `how` is an integer specifying the type of shutdown corresponding to the (r, w) arguments in the model shutdown(). If `how` is set to `SHUT_RD` then the read half of the connection is to be shut down, corresponding to a shutdown(fd, $\mathbf{T}, \mathbf{F}$) 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 shutdown(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 shutdown(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.

15.27.4 Model details

The following errors are not modelled:

- `EINVAL` signifies that the `how` argument is invalid. In the model the `how` argument is represented by the two boolean flags r and w which guarantees that the only values allowed are $(\mathbf{T}, \mathbf{T})$, $(\mathbf{T}, \mathbf{F})$, $(\mathbf{F}, \mathbf{T})$, 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 $\mathsf{shutdown}(\mathsf{fd}, \mathbf{F}, \mathbf{F})$ 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.

15.27.5 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

15.27.6 Rules

shutdown_1 **tcp: fast succeed** **Shut down read or write half of TCP connection**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\ socks := socks \oplus [(sid, sock)]|\!\rangle \xrightarrow{tid\cdot \mathsf{shutdown}(fd, r, w)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{sched\_timer}});\ socks := socks \oplus [(sid, sock')]|\!\rangle$$

$sock = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore, pr) \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$pr = \text{TCP\_PROTO}\ tcp\_sock \wedge$
if $\text{bsd\_arch}\ h.arch \wedge tcp\_sock.st \in \{\text{CLOSED}; \text{LISTEN}\} \wedge w$ **then**

$$\begin{array}{l}
\textbf{let}\ sock'' = (\text{tcp_close}\ h.arch\ sock)\ \textbf{in}\\
sock' = sock''\ \langle\!|\ cantsndmore := (w \vee cantsndmore);\\
\quad cantrcvmore := (r \vee cantrcvmore);\\
\quad pr := \text{TCP_PROTO}(\text{tcp_sock_of}\ sock''\\
\qquad \langle\!|\ cb\ :\hat{=} (\lambda cb.cb\ \langle\!|\ bsd_cantconnect := \mathbf{T}|\!\rangle);\\
\qquad\ \ lis := *|\!\rangle)\\
\quad |\!\rangle
\end{array}$$

else

$$\begin{array}{l}
(\neg\,\text{bsd_arch}\ h.arch \implies \exists i_1\ p_1\ i_2\ p_2.tcp_sock.st = \text{ESTABLISHED} \wedge is_1 = \uparrow i_1 \wedge\\
\qquad ps_1 = \uparrow p_1 \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2 \wedge tcp_sock.lis = *) \wedge\\
pr' = \text{TCP_PROTO}(tcp_sock\ \langle\!|\ rcvq\ :\hat{=} [\,]\ \textbf{onlywhen}\ r;\\
\qquad cb\ :\hat{=} (\lambda cb.cb\ \langle\!|\\
\qquad\quad tf_shouldacknow\ :\hat{=}\ \mathbf{T}\ \textbf{onlywhen}\ w|\!\rangle)|\!\rangle) \wedge\\
sock' = \text{SOCK}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, w \vee cantsndmore, r \vee cantrcvmore, pr')
\end{array}$$

Description

From thread tid, which is in the RUN state, a $\mathsf{shutdown}(fd, r, w)$ 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 $tid\cdot\mathsf{shutdown}(fd, r, w)$ transition is made, leaving the thread in state $\text{RET}(\text{OK}())$. 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 (p121). Note that testing has shown the socket's listen queue is not always set to $*$ after a shutdown() call. The precise condition for this being done needs to be investigated.

shutdown_2 **udp: fast succeed** **Shutdown UDP socket for reading, writing, or both**

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

$$\xrightarrow{tid\cdot\mathsf{shutdown}(fd, r, w)} h \;\langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{sched\_timer}});\; socks := socks \oplus [(sid, sock\;\langle\!|cantrcvmore :=(r \vee cantrcvmore);\; cantsndmore :=(w \vee cantsndmore);\; pr := \mathrm{UDP\_PROTO}(udp\_pr)|\!\rangle)]|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\mathrm{FT\_}\textsc{Socket}(sid), ff) \wedge$
$(\mathrm{linux\_arch}\; h.arch \implies sock.is_2 \neq *)$

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 $tid\cdot$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**

$$h \;\langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d)|\!\rangle$$

$$\xrightarrow{tid\cdot\mathsf{shutdown}(fd, r, w)} h \;\langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{FAIL\ ENOTCONN}))_{\mathrm{sched\_timer}})|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\mathrm{FT\_}\textsc{Socket}(sid), ff) \wedge$
$\mathrm{TCP\_PROTO}(tcp\_sock) = (h.socks[sid]).pr \wedge$
$tcp\_sock.st \neq \mathrm{ESTABLISHED} \wedge$

$\neg(\text{bsd\_arch } h.arch)$

Description

From thread *tid*, which is in the Run state, a $\mathsf{shutdown}(fd, r, w)$ 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 $tid{\cdot}\mathsf{shutdown}(fd, r, w)$ 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**

$$
\begin{array}{l}
h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, sock \ \langle\!|is_2 := *; pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle\\
\xrightarrow{tid\cdot\mathsf{shutdown}(fd,r,w)} \ h \ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL ENOTCONN}))_{\text{sched\_timer}});\\
\qquad socks := socks \oplus\\
\qquad\quad [(sid, sock \ \langle\!|is_2 := *;\\
\qquad cantsndmore := (w \vee sock.cantsndmore);\\
\qquad cantrcvmore := (r \vee sock.cantrcvmore);\\
\qquad pr := \text{UDP\_PROTO}(udp)|\!\rangle)]|\!\rangle
\end{array}
$$

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

Description

On 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 $\mathsf{shutdown}(fd, r, w)$ call is made, and fails with an ENOTCONN error.

A $tid{\cdot}\mathsf{shutdown}(fd, r, w)$ 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*.

15.28 sockatmark() (TCP only)

$\mathsf{sockatmark} : \mathsf{fd} \to \mathsf{bool}$

A call to sockatmark(fd) returns a bool specifying whether or not a socket is at the urgent mark. Here fd is a file descriptor referring to a socket.

If fd refers to a TCP socket then the call will succeed, returning **T** if that socket is at the urgent mark, and **F** if it is not.

If fd refers to a UDP socket then on FreeBSD the call will return **F** and on all other architectures it will fail with an EINVAL error: there is no concept of urgent data for UDP so calling sockatmark() does not make sense.

15.28.1 Errors

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

EINVAL	Calling sockatmark() on a UDP socket does not make sense.
EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.

15.28.2 Common cases

sockatmark_1; *return_1*

15.28.3 API

Posix:	`int sockatmark(int s);`
FreeBSD:	`int ioctl(int d, unsigned long request, int* argp);`
Linux:	`int ioctl(int d, int request, int* argp);`
WinXP:	`int ioctlsocket(SOCKET s, long cmd, u_long* argp);`

In the Posix interface:

- `s` is a file descriptor referring to a socket. This corresponds to the fd argument of the model sockatmark().
- the returned `int` is either `0` or `1` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. If the return value is `1` then the socket is at the urgent mark corresponding to a return value of **T** in the model sockatmark(); if the return value is `0` then the socket is not at the urgent mark, corresponding to a return value of **F** in the model.

The FreeBSD, Linux, and WinXP interfaces are significantly different: to check whether or not a socket is at the urgent mark, the `ioctl()` function must be used. In the FreeBSD interface:

- `d` is a file descriptor referring to a socket, corresponding to the fd argument of the model sockatmark().
- `request` selects which control function is to be performed. For sockatmark(), the request is `SIOCATMARK`.
- `argp` is a pointer to a location to store the result of the call in. If the socket is at the urgent mark then `1` will be in the location pointed to by `argp` upon return, corresponding to a return value of **T** in the model sockatmark(); if the socket is not at the urgent mark, then `argp` will contain the value `0`, corresponding to a return value of **F** in the model.
- the returned `int` is either `0` to indicate success or `-1` to indicate an error, in which case the error code is in `errno`. On WinXP an error is indicated by a return value of `SOCKET_ERROR`, not `-1`, with the actual error code available through a call to `WSAGetLastError()`.

The Linux and WinXP interfaces are similar.

15.28.4 Model details

The following errors are not modelled:

- On FreeBSD, Linux, and WinXP, EFAULT can be returned if the `argp` parameter points to memory not in a valid part of the process address space. This is an artefact of the C interface to `ioctl()` that is excluded by the clean interface used in the model sockatmark().
- On FreeBSD and Linux, EINVAL can be returned if `request` is not a valid request. The model sockatmark() is implemented using the `SIOCATMARK` request which is valid.

- ENOTTY is possible when making an *ioctl*() call but is not modelled.
- `WSAEINPROGRESS` is WinXP-specific and described in the MSDN page as "A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function". This is not modelled here.

15.28.5 Summary

sockatmark_1	**tcp: fast succeed**	Successfully return whether or not a TCP socket is at the urgent mark
sockatmark_2	**udp: rc**	Fail with EINVAL: calling sockatmark() on a UDP socket does not make sense

15.28.6 Rules

sockatmark_1 **tcp: fast succeed** **Successfully return whether or not a TCP socket is at the urgent mark**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d)|\!\rangle \xrightarrow{tid\cdot\mathsf{sockatmark}(fd)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK } b))_{\text{sched\_timer}})|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
$h.socks[sid] = \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(\text{ESTABLISHED}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$b = (rcvurp = \uparrow 0)$

Description

From thread *tid*, which is in the Run state, a sockatmark(*fd*) call is made. *fd* refers to a TCP socket identified by *sid* which is in the ESTABLISHED state and has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$. The call succeeds, returning **T** if the socket is at the urgent mark: $rcvurp = \uparrow 0$; or **F** otherwise.

A *tid*·sockatmark(*fd*) transition is made, leaving the thread state Ret(OK *b*) where *b* is a boolean: **T** or **F** as above.

sockatmark_2 **udp: rc** **Fail with EINVAL: calling** sockatmark() **on a UDP socket does not make sense**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d)|\!\rangle \xrightarrow{tid\cdot\mathsf{sockatmark}(fd)} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(ret))_{\text{sched\_timer}})|\!\rangle$$

$\text{proto\_of}(h.socks[sid]).pr = \text{PROTO\_UDP} \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
$\mathbf{if}\ \text{bsd\_arch}\ h.arch\ \mathbf{then}\ rc = \textsc{fast succeed} \wedge ret = \text{OK}(\mathbf{F})$
$\mathbf{else}\ rc = \textsc{fast fail} \wedge ret = \text{FAIL EINVAL}$

Description

Consider a UDP socket *sid* referenced by *fd*. From thread *tid*, which is in the Run state, a sockatmark(*fd*) call is made. On FreeBSD the call succeeds, returning **F**; on Linux and WinXP the call fails with an EINVAL error.

A *tid*·sockatmark(*fd*) transition is made, leaving the thread state Ret(OK(**F**)) on FreeBSD, and in state Ret(FAIL EINVAL) on Linux and WinXP.

Variations

Posix	As above: the call succeeds, returning **F**.
FreeBSD	As above: the call succeeds, returning **F**.
Linux	As above: the call fails with an EINVAL error.
WinXP	As above: the call fails with an EINVAL error.

15.29 socket() (TCP and UDP)

socket : *sock_type* → fd

A call to socket(*type*) creates a new socket. Here *type* is the type of socket to create: SOCK_STREAM for TCP and SOCK_DGRAM for UDP. The returned fd is the file descriptor of the new socket.

15.29.1 Errors

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

EMFILE	No more file descriptors for this process.
ENOBUFS	Out of resources.
ENOMEM	Out of resources.
ENFILE	Out of resources.

15.29.2 Common cases

TCP: *socket_1*; *return_1*; *connect_1*; ... UDP: *socket_1*; *return_1*; *bind_1*; *return_1*; *send_9*; ...

15.29.3 API

Posix:	`int socket(int domain, int type, int protocol);`
FreeBSD:	`int socket(int domain, int type, int protocol);`
Linux:	`int socket(int doamin, int type, int protocol);`
WinXP:	`SOCKET socket(int af, int type, int protocol);`

In the Posix interface:

- `domain` specifies the communication domain in which the socket is to be created, specifying the protocol family to be used. Only IPv4 sockets are modelled here, so `domain` is set to `AF_INET` or `PF_INET`.
- `type` specifies the communication semantics: `SOCK_STREAM` provides sequenced, reliable, two-way, connection-based byte streams; `SOCK_DGRAM` supports datagrams (connectionless, unreliable messages of a fixed maximum length). This corresponds to the *sock_type* argument of the model socket().
- `protocol` specifies the particular protocol to be used for the socket. A `protocol` of `0` requests to use the default for the appropriate socket `type`: TCP for `SOCK_STREAM` and UDP for `SOCK_DGRAM`. Alternatively a specific protocol number can be used: `6` for TCP and `17` for UDP. In the model, SOCK_STREAM refers to a TCP socket and SOCK_DGRAM to a UDP socket so the `protocol` argument is not necessary.

A call to socket(SOCK_STREAM) in the model interface, would be a `socket(AF_INET,SOCK_STREAM,0)` call in Posix; a call to socket(SOCK_DGRAM) in the model interface would be a `socket(AF_INET,SOCK_DGRAM,0)` call in Posix.

The FreeBSD, Linux and WinXP interfaces are similar modulo argument renaming, except where noted above.

15.29.4 Model details

The following errors are not modelled:

- In Posix and on Linux, `EACCES` specifies that the process does not have appropriate privileges. We do not model a privilege state in which socket creation would be disallowed.
- In Posix and on Linux, `EAFNOSUPPORT`, specifies that the implementation does not support the address `domain`. FreeBSD, Linux, and WinXP all support `AF_INET` sockets.
- On Linux, `EINVAL` means unknown protocol, or protocol domain not available. Both TCP and UDP are known protocols for Linux, and `AF_INET` is a known domain on Linux.
- In Posix and on Linux, `EPROTONOTSUPPORT` specifies that the protocol is not supported by the address family, or the protocol is not supported by the implementation. FreeBSD, Linux, and WinXP all support the TCP and UDP protocols.
- In Posix, `EPROTOTYPE` signifies that the socket type is not supported by the protocol. Both `SOCK_STREAM` and `SOCK_DGRAM` are supported by TCP and UDP respectively.
- On WinXP, `WSAESOCKTNOSUPPORT` means the specified socket type is not supported in this address family. The `AF_INET` family supports both `SOCK_STREAM` and `SOCK_DGRAM` sockets.

The `AF_INET6`, `AF_LOCAL`, `AF_ROUTE`, and `AF_KEY` address families; `SOCK_RAW` socket type; and all protocols other than TCP and UDP are not modelled.

15.29.5 Summary

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

15.29.6 Rules

socket_1 **all: fast succeed** **Successfully return a new file descriptor for a fresh socket**

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d);\\
\quad fds := fds;\\
\quad files := files;\\
\quad socks := socks|\!\rangle\\
\xrightarrow{tid\cdot(\mathsf{socket}(socktype))}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}\ fd))_{\text{sched\_timer}});\\
\quad fds := fds';\\
\quad files := files \oplus [(fid, \text{FILE}(\text{FT\_SOCKET}(sid), \text{ff\_default}))];\\
\quad socks := socks \oplus [(sid, sock)]|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathbf{card}(\mathbf{dom}(fds)) < \text{OPEN\_MAX} \wedge\\
fid \notin (\mathbf{dom}(files)) \wedge\\
sid \notin (\mathbf{dom}(socks)) \wedge\\
\text{nextfd}\ h.arch\ fds\ fd \wedge\\
fds' = fds \oplus (fd, fid) \wedge\\
(\mathbf{case}\ socktype\ \mathbf{of}\\
\quad \text{SOCK\_DGRAM} \rightarrow (sock =\\
\qquad \text{SOCK}(\uparrow fid, \text{sf\_default}\ h.arch\ socktype, *, *, *, *, *, \mathbf{F}, \mathbf{F}, \text{UDP\_Sock}([\,]))) \parallel\\
\quad \text{SOCK\_STREAM} \rightarrow (sock =\\
\qquad \text{SOCK}(\uparrow fid, \text{sf\_default}\ h.arch\ socktype, *, *, *, *, *, \mathbf{F}, \mathbf{F},
\end{array}
$$

$\text{TCP\_Sock}(\text{CLOSED}, \text{initial\_cb}, *, [\,], *, [\,], *, \text{NO\_OOBDATA}))))$

Description

From thread *tid*, which is in the RUN state, a socket(*socktype*) call is made. The number of open file descriptors is less than the maximum permitted, OPEN_MAX.

If *socktype* = SOCK_STREAM then a new TCP socket *sock* is created, in the CLOSED state, with initial_cb (p101) as its control block, and all other fields uninitialised; if *socktype* = SOCK_DGRAM then a new, 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 (p**??**)) to point to the open file description. The host's finite map of sockets is updated to include an entry mapping the socket identifier *sid* to the socket; its finite map of file descriptions is updated to add an entry mapping the file descriptor *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(OK*fd*) to return the new file descriptor.

socket_2 <u>**all: fast fail**</u> **Fail with EMFILE: out of file descriptors for this process**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle \xrightarrow{tid\cdot(\mathsf{socket}(s))} h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EMFILE}))_{\text{sched\_timer}})|\!\rangle$$

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

Description

From thread *tid*, which is in the RUN state, a socket(s) call is made. The number of open file descriptors is greater than the maximum allowed number, OPEN_MAX, and so the call fails with an EMFILE error.

A *tid*·socket(s) transition is made, leaving the thread state RET(FAIL EMFILE).

15.30 Miscellaneous (TCP and UDP)

This section collects the remaining Sockets API rules:

- The rule *return_1* characterising how the the results of system calls are returned to the caller, with transitions from the thread state $(\text{RET}\ v)_d$.
- Rules *badf_1* and *notsock_1* deal with all the Sockets API calls that take a file descriptor argument, dealing uniformly with the error cases in which that file descriptor is not valid or does not refer to a socket.
- Rule *intr_1* applies to all the thread states for blocked calls, ACCEPT2(sid) etc., characterising the behaviour in the case where the call is interrupted by a signal.
- Rules *resourcefail_1* and *resourcefail_2* deal with the cases where calls fail due to a lack of system resources.

15.30.1 Errors

Common errors.

EBADF	The file descriptor passed is not a valid file descriptor.
ENOTSOCK	The file descriptor passed does not refer to a socket.
EINTR	The system was interrupted by a caught signal.

ENOMEM	Out of resources.
ENOBUFS	Out of resources.
ENFILE	Out of resources.

15.30.2 Summary

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

15.30.3 Rules

return_1 **all: misc nonurgent** **Return result of system call to caller**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RET}\ v)_d)|\!\rangle \xrightarrow{tid \cdot v} h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_{\mathrm{never\_timer}})|\!\rangle$$

T

Description

A system call from thread *tid* has completed, leaving the thread state $(\mathrm{RET}\ v)_d$. 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 $(\mathrm{RUN})_{\mathrm{never\_timer}}$.

badf_1 **all: fast fail** **Fail with EBADF: not a valid file descriptor**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RUN})_d)|\!\rangle \xrightarrow{tid \cdot opn} h\ \langle\!|ts := ts \oplus (tid \mapsto (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{sched\_timer}})|\!\rangle$$

$\mathrm{fd\_op}\ fd\ opn\ \wedge$
$fd \notin \mathbf{dom}(h.fds)\ \wedge$
$(\mathbf{if}\ \ \mathrm{windows\_arch}\ h.arch\ \ \mathbf{then}\ \ e = \mathrm{ENOTSOCK}\ \ \mathbf{else}\ \ e = \mathrm{EBADF})$

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 $tid \cdot opn$ 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 (p35).

Variations

FreeBSD	As above: the call fails with an EBADF error.
Linux	As above: the call fails with an EBADF error.
WinXP	As above: the call fails with an ENOTSOCK error.

notsock_1 **all: fast fail** **Fail with ENOTSOCK: file descriptor not a valid socket**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d)|\!\rangle \xrightarrow{tid \cdot opn} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL ENOTSOCK}))_{\text{sched\_timer}})|\!\rangle$$

fd_sockop $fd\ opn \wedge$
$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(ft, ff) \wedge$
$\neg(\exists sid.ft = \text{FT\_}\textsc{Socket}(sid))$

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 $\textsc{File}(ft, ff)$ that does not refer to a socket. The call fails with an ENOTSOCK error.

A $tid \cdot opn$ 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 (p35).

intr_1 **all: slow nonurgent fail** **Fail with EINTR: blocked system call interrupted by signal**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (st)_d)|\!\rangle \xrightarrow{\tau} h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINTR}))_{\text{sched\_timer}})|\!\rangle$$

$sock = (h.socks[sid]) \wedge$
$(st = \textsc{Close2}(sid) \vee$
$st = \textsc{Connect2}(sid) \vee$
$st = \textsc{Recv2}(sid, n, opts) \vee$
$st = \textsc{Send2}(sid, addr, str, opts) \vee$
$st = \text{P}\textsc{Select2}(readfds, writefds, exceptfds) \vee$
$st = \textsc{Accept2}(sid))$

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**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RUN})_d)|\!\rangle \quad \xrightarrow{tid \cdot call} \quad h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ e))_{\text{sched\_timer}})|\!\rangle$$

$\neg$ INFINITE_RESOURCES $\wedge$
$fd \ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \text{FILE}(\text{FT\_SOCKET}(sid), ff) \wedge$
$sock = (h.socks[sid]) \wedge$
$((call = \mathsf{socket}(socktype) \wedge e \ \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\}) \vee$
$(call = \mathsf{bind}(fd, is_1, ps_1) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{connect}(fd, i_2, \uparrow p_2) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{listen}(fd, n) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{recv}(fd, n, opts) \wedge e \ \in \{\text{ENOMEM}; \text{ENOBUFS}\}) \vee$
$(call = \mathsf{getsockname}(fd) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{getpeername}(fd) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{shutdown}(fd, r, w) \wedge e = \text{ENOBUFS}) \vee$
$(call = \mathsf{accept}(fd) \wedge e \ \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\}$
$\wedge$ proto_of $\ sock.pr = \text{PROTO\_TCP}))$

Description

Thread *tid* performs a socket(), bind(), connect(), listen(), recv(), getsockname(), getpeername(), shutdown() or accept() system call on socket *sid*, referred to by *fd*, when insufficient system-wide resources are available to complete the request. Return a failure of ENFILE, ENOBUFS or ENOMEM immediately to the calling thread.

This rule applies only when it is assumed that the host being modelled does not have INFINITE_RESOURCES, i.e. the host does not have unlimited memory, mbufs, file descriptors, etc.

Model details

The modelling of failure is deliberately non-deterministic because the cause of errors such as ENFILE are determined by more than is modelled in this specification. In order to be more precise, the model would need to describe the whole system to determine when such error conditions could and should arise.

resourcefail_2 **all: slow nonurgent badfail** **Fail with ENFILE, ENOBUFS or ENOMEM: from a blocked state with out of resources**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (t)_d)|\!\rangle \quad \xrightarrow{\tau} \quad h\ \langle\!|ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL}\ e))_{\text{sched\_timer}})|\!\rangle$$

$\neg$ INFINITE_RESOURCES $\wedge$
$sock = (h.socks[sid]) \wedge$
$((t = \text{ACCEPT2}(sid) \wedge e \ \in \{\text{ENFILE}; \text{ENOBUFS}; \text{ENOMEM}\}) \vee$
$(t = \text{CONNECT2}(sid) \wedge e = \text{ENOBUFS}) \vee$
$(t = \text{RECV2}(sid, n, opts) \wedge e \ \in \{\text{ENOBUFS}; \text{ENOMEM}\}))$

Description

If thread *tid* of host *h* is in state ACCEPT2(*sid*), CONNECT2(*sid*) or 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 16

Host LTS: TCP Input Processing

16.1 Input Processing (TCP only)

These rules deal with the processing of TCP segments from the host's input queue. The most important are *deliver_in_1*, *deliver_in_2*, and *deliver_in_3*.

deliver_in_1 deals with a passive open: a socket in LISTEN state that receives a *SYN* and sends a *SYN*, *ACK*.

deliver_in_2 deals with the completion of an active open: a socket in SYN_SENT state (that has previously sent a *SYN* with the *connect_1* rule) that receives a *SYN*, *ACK* and sends an *ACK*. It also deals with simultaneous opens.

deliver_in_3 deals with the common cases of TCP data exchange and connection close: sockets in connected states that receive data, *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.

16.1.1 Summary

deliver_in_1	**tcp: network nonurgent**	Passive open: receive SYN, send SYN,ACK
deliver_in_1b	**tcp: network nonurgent**	For a listening socket, receive and drop a bad datagram and either generate a RST segment or ignore it. Drop the incoming segment if the socket's queue of incomplete connections is full.
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_2a	**tcp: network nonurgent**	Receive bad or boring datagram and RST or ignore for SYN_SENT socket
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_really		*deliver_in_3* data 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_3a	**tcp: network nonurgent**	Receive data with invalid checksum or offset
deliver_in_3b	**tcp: network nonurgent**	Receive data after process has gone away
deliver_in_3c	**tcp: network nonurgent**	Receive stupid ACK or LAND DoS in SYN_RECEIVED state
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_6	**tcp: network nonurgent**	Receive and drop (silently) a sane segment that matches a CLOSED 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

16.1.2 Rules

deliver_in_1 **tcp: network nonurgent** **Passive open: receive SYN, send SYN,ACK**

$$h\ \langle\!\langle socks := socks \oplus [(sid, sock)];$$
$$iq := iq;$$
$$oq := oq \rangle\!\rangle$$

$\xrightarrow{\tau}$

$$h\ \langle\!\langle socks := socks' \oplus$$
(* Listening socket *)
$$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$$
$$\text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis', [\,], *, [\,], *, \text{NO\_OOBDATA})));$$
(* New socket formed by the incoming SYN *)
$$(sid', \textsc{Sock}(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, cantsndmore, cantrcvmore,$$
$$\text{TCP\_Sock}(\text{SYN\_RECEIVED}, cb'', *, [\,], *, [\,], *, \text{NO\_OOBDATA})))];$$
$$iq := iq';$$
$$oq := oq' \rangle\!\rangle$$

(* **Summary:** A host h with listening socket $sock$ referenced by index sid receives a valid and well-formed SYN segment seg addressed to socket $sock$. A new socket in the SYN_RECEIVED state is constructed, referenced by $sid' (\neq sid)$, is added to the queue of incomplete incoming connection attempts q, and a SYN,ACK segment is generated in reply with some field values being chosen or negotiated. The reply segment is finally queued on the host's output queue for transmission, ignoring any errors upon queueing failure. *)

$sid \notin (\mathbf{dom}(socks)) \wedge$
$sid' \notin (\mathbf{dom}(socks)) \wedge$
$sid \neq sid' \wedge$

(* Take TCP segment seg from the head of the host's input queue *)
$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$

(* The segment must be of an acceptable form *)
(* Note: some segment fields are ignored during TCP connection establishment and as such may contain arbitrary values. These are equal to the identifiers postfixed with *_discard* below, which are otherwise unconstrained. *)
$(\exists win\_\ ws\_\ mss\_\ PSH\_discard\ URG\_discard\ FIN\_discard\ urp\_discard\ data\_discard\ ack\_discard.$
$seg =$
$\langle\!\langle is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq : \text{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack\_discard : \text{tcp\_seq\_local});$

$URG := URG\_discard;$
$ACK := \mathbf{F};$ (* ACK must be **F** in a SYN segment *)
$PSH := PSH\_discard;$
$RST := \mathbf{F};$ (* Valid SYN segments never have RST set *)
$SYN := \mathbf{T};$ (* Is a SYN segment! *)
$FIN := FIN\_discard;$
$win := win\_;$
$ws := ws\_;$
$urp := urp\_discard;$
$mss := mss\_;$
$ts := ts;$
$data := data\_discard$
$\rrbracket \wedge$

(* Equality of some type casts *)
$\mathbf{w2n}\ win\_ = win \wedge$
$\mathbf{option\_map}\ \mathbf{ord}\ ws\_ = ws \wedge$
$\mathbf{option\_map}\ \mathbf{w2n}\ mss\_ = mss$
$) \wedge$

(* 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 *)
$i_1 \in \text{local\_ips}\ h.ifds \wedge$
$\neg(\text{is\_broadormulticast}\ h.ifds\ i_1) \wedge$
$\neg(\text{is\_broadormulticast}\ h.ifds\ i_2) \wedge$

(* Find the socket *sock* that has the best match for the address quad in segment *seg*, see tcp_socket_best_match (p86). Socket *sock* must have a form matching the patten $\text{Sock}(\ldots)$. *)
$\text{tcp\_socket\_best\_match}\ socks(sid, sock)seg\ h.arch \wedge$
$sock = \text{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
$\qquad \text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \text{NO\_OOBDATA})) \wedge$

(* 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 $*$. *)
$((is_2 = * \wedge ps_2 = *) \vee$
$(\text{bsd\_arch}\ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

(* 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* (p285) and not here. *)
$(\mathbf{case}\ is_1\ \mathbf{of}\ \uparrow i1' \rightarrow i1' = i_1 \parallel * \rightarrow \mathbf{T}) \wedge$
$\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$

(* 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. *)
$\neg(\exists(sid, sock) :: socks.$
$\quad \exists tcp\_sock.$
$\quad sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\quad tcp\_sock.st = \text{TIME\_WAIT} \wedge$
$\quad sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2 \wedge$
$\quad seq \leq tcp\_sock.cb.rcv\_nxt) \wedge$

(* 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. *)

$socks' = \$o\_f(\lambda sock.$
  if $\exists tcp\_sock.sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
    $tcp\_sock.st = \mathrm{TIME\_WAIT} \wedge$
    $sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge$
    $sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2$
  then
    tcp_close *h.arch sock*
  else
    sock
$)socks \wedge$

(* Accept the new connection attempt to the incomplete connection queue if the queue of completed (established) connections is not already full *)

accept_incoming_q0 *lis* $\mathbf{T} \wedge$

(* 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. *)

(**choose** *drop* :: drop_from_q0 *lis*.
  if *drop* **then**
    $\exists q0L\ sid''\ q0R.$
      $lis.q_0 = q0L\ @\ (sid'' :: q0R) \wedge$
      $q_0' = q0L\ @\ q0R$
  else
    $q_0' = lis.q_0$
$) \wedge$

(* Put the new incomplete connection on the (possibly pruned) incomplete connections queue. *)

$lis' = lis\ \langle\!|\ q_0 := sid' :: q_0' |\!\rangle \wedge$

(* Create a SYN,ACK segment in reply: *)

(* The maximum segment size of the outgoing SYN,ACK reply segment must be in range, i.e., less than the maximum IP segment size minus the space consumed by IP and TCP headers. This is deliberately non-deterministic: an implementation would query the interface's MTU and subtract the header space required. *)

$advmss\ \in \{n \mid n \geq 1 \wedge n \leq (65535 - 40)\} \wedge$

(* Be non-deterministic in deciding whether to transmit a maximum segment size option. A host either supports the maximum segment size option or not – here the specfication permits either sending the option or not, but if the option is sent it must contain the advertised mss chosen previously by the host. This captures all acceptable behaviour. *)

$advmss'\ \in \{*; \uparrow advmss\} \wedge$

(* If a timestamp option was present in the received segment and a non-deterministic choice is made to do timestamping on this connection (i.e., the host supports timestamping), then timestamping is being used for this connection. Otherwise, timestamping is not used because one or both hosts do not support it. A real host would either do timestamping or not depending on its configuration. Here all acceptable behaviour must be permitted. *)

$tf\_rcvd\_tstmp' =$ **is_some** $ts \wedge$
(**choose** *want_tstmp* :: $\{\mathbf{F}; \mathbf{T}\}$.
  $tf\_doing\_tstmp' = (tf\_rcvd\_tstmp' \wedge want\_tstmp)$
$) \wedge$

(* Lookup the bandwidth delay product from the route metric cache and calculate the size of the receive and send buffers, the maximum segment size and the initial congestion window. *)

$bw\_delay\_product\_for\_rt = * \wedge$
$(rcvbufsize', sndbufsize', t\_maxseg', snd\_cwnd') =$

$$\text{calculate\_buf\_sizes}\ advmss\ mss\ bw\_delay\_product\_for\_rt(\text{is\_localnet}\ h.ifds\ i_2)$$
$$(sf.n(\text{SO\_RCVBUF}))(sf.n(\text{SO\_SNDBUF}))tf\_doing\_tstmp'\ h.arch\ \wedge$$

(* Store the new receive and send buffer sizes *)
$sf' = sf\ (\!| n := \text{funupd\_list}\ sf.n[(\text{SO\_RCVBUF}, rcvbufsize'); (\text{SO\_SNDBUF}, sndbufsize')] |\!) \wedge$

(* Non-deterministically choose to do window scaling (i.e., choose whether this host supports window scaling or not). Do window scaling on the new connection if the received SYN segment contained a window scaling option and this host supports it. A real host would either be configured to do window scaling or not (provided it supported window scaling). Here all acceptable behaviour must be permitted. *)
$req\_ws \in \{\mathbf{F}; \mathbf{T}\} \wedge$
$tf\_doing\_ws' = (req\_ws \wedge \textbf{is\_some}\ ws) \wedge$
(**if** tf_doing_ws' **then** (* Doing window scaling *)
(* Constrain the receive scale to be within the correct range and the send scale to be that received from the remote host *)
$rcv\_scale' \in \{n \mid n \geq 0 \wedge n \leq \text{TCP\_MAXWINSCALE}\} \wedge snd\_scale' = \textbf{option\_case}\ 0\ \mathbf{I}\ ws$
else
(* Otherwise, turn off scaling *)
$rcv\_scale' = 0 \wedge snd\_scale' = 0) \wedge$

(* Constrain the receive window for the new connection – this is advertised in the *SYN*,*ACK* reply. No scaling is performed here as scaling is not applied to segments containing a valid *SYN* since the support for window scaling has not been fully negotitated yet! *)

$$rcv\_window \in \{n \mid n \geq 0 \wedge n \leq \text{TCP\_MAXWIN} \wedge n \leq sf.n(\text{SO\_RCVBUF})\} \wedge$$

(* Time the SYN,ACK reply segment. This is a new connection thus no previous timers can be running. *)
(**let** $t\_rttseg' = \uparrow(\text{ticks\_of}\ h.ticks, cb.snd\_nxt)$ **in**

(* Initial sequence number of *SYN*,*ACK* reply segment is unconstrained. *)
$iss \in \{n \mid \mathbf{T}\} \wedge$
(* The *ack* value in the reply segment must acknowledge the remote host's initial *SYN*. *)
let $ack' = seq + 1$ **in**

(* Update the new connection's control block in light of above. *)
$cb' = cb\ (\!|$

$tt\_keep := \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_IDLE}})$;
$tt\_rexmt := \text{start\_tt\_rexmt}\ h.arch\ 0\ \mathbf{F}\ cb.t\_rttinf$;
$iss := iss$;
$irs := seq$;
$rcv\_wnd := rcv\_window$;
$tf\_rxwin0sent := (rcv\_window = 0)$;
$rcv\_adv := ack' + rcv\_window$;
$rcv\_nxt := ack'$;
$snd\_una := iss$;
$snd\_max := iss + 1$; (* *SYN* consumes one-byte of sequence space *)
$snd\_nxt := iss + 1$; (* *SYN* consumes one-byte of sequence space *)
$snd\_cwnd := snd\_cwnd'$;
$rcv\_up := seq + 1$; (* Pull along with left edge of unused window *)
$t\_maxseg := t\_maxseg'$; (* The negotiated mss, with options removed *)
$t_{advmss} := advmss'$; (* Remember the mss advertised (if any) by this socket in case the *SYN* segment is retransmitted *)
$rcv\_scale := rcv\_scale'$;
$snd\_scale := snd\_scale'$;
$tf\_doing\_ws := tf\_doing\_ws'$;
$ts\_recent :=$ **case** ts **of**

$$* \rightarrow cb.ts\_recent \parallel$$
$$\uparrow(ts\_val, ts\_ecr) \rightarrow (ts\_val)^{\text{TimeWindow}}_{\text{kern\_timer dtsinval}};$$
$last\_ack\_sent := ack'$;
$t\_rttseg := t\_rttseg'$;
$tf\_req\_tstmp := tf\_doing\_tstmp'$;
$tf\_doing\_tstmp := tf\_doing\_tstmp'$
$\rrbracket) \wedge$

(* Construct the SYN,ACK segment using the values stored in the updated control block for the new connection. See make_syn_ack_segment (p107). *)
choose seg' :: make_syn_ack_segment $cb'(i_1, i_2, p_1, p_2)$(ticks_of $h.ticks$).

(* Add the SYN,ACK reply segment to the host's output queue, ignoring failure. Constrain the new connection's initial control block cb to have just the right values in case queueing of the segment fails (perhaps due to a routing failure) and some control block state has to be rolled back. See rollback_tcp_output (p117) and enqueue_or_fail (p118) for more detail. *)
enqueue_or_fail **T** $h.arch$ $h.rttab$ $h.ifds$[TCP seg']oq
$(cb$
$\llbracket$ $snd\_nxt := iss$; (* If queueing fails, need to retransmit the SYN *)
$snd\_max := iss$; (* If queueing fails, need to retransmit the SYN *)
$t\_maxseg := t\_maxseg'$;
$last\_ack\_sent :=$ tcp_seq_foreign $0w$;
$rcv\_adv :=$ tcp_seq_foreign $0w$
$\rrbracket)cb'(cb'', oq')$

Model details

During TCP connection establishment, BSD uses syn-caches and syn-buckets to protect against some types of denial-of-service attack. These techniques delay the memory allocation for a socket's data structures until connection establishment is complete. They are not modelled directly in this specification, which instead favours the use of the full socket structure for clarity. The behaviour is observationally equivalent provided correct bounds are applied to the lengths of the incoming connection queues.

When a socket completes connection establishment, i.e., enters the ESTABLISHED state, BSD updates the socket's control block t_maxseg field to the minimum of the maximum segment size it advertised in the emitted SYN,ACK segment and that received in the SYN segment from the remote end. This update is later than perhaps it need be. This model updates the t_maxseg at the moment both the maximum segment values are known. As a consequence the initial maximum segment value advertised by the host must be stored just in case the SYN,ACK segment need be retransmitted.

Variations

FreeBSD	On FreeBSD, the 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_1b **tcp: network nonurgent** **For a listening socket, receive and drop a bad datagram and either generate a RST segment or ignore it. Drop the incoming segment if the socket's queue of**

incomplete connections is full.

h ⦇$socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ h ⦇$socks := socks \oplus [(sid, sock)];$
$iq := iq;$ $iq := iq';$
$oq := oq;$ $oq := oq';$
$bndlm := bndlm$⦈ $bndlm := bndlm'$⦈

(* **Summary**: A host h with listening socket *sock* referenced by index *sid* receives a segment *seg* addressed to socket *sock*. The segment either contains an invalid combination of the *SYN* and *ACK* flags, is a forged segment trying to force the listening socket *sock* to connect to itself, or the new incomplete connection can not be added to the queue of incomplete connections because the completed connections queue is full. The segment is dropped. If the segment had the *ACK* flag set and not *SYN*, a RST segment is generated and added to the host's output queue *oq* for transmission. *)

(* Take TCP segment *seg* from the head of the host's input queue *)
dequeue_iq($iq, iq', \uparrow$(TCP seg)) $\wedge$

(* The segment must be of an acceptable form *)
(* Note: some segment fields are ignored during TCP connection establishment and as such may contain arbitrary values. These are equal to the identifiers postfixed with *_discard* below, which are otherwise unconstrained. *)
($\exists$*seq_discard ack_discard URG_discard PSH_discard FIN_discard*
win_discard ws_discard urp_discard mss_discard ts_discard data_discard.
seg =⦇
$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
seq := *tcp_seq_flip_sense*(*seq_discard* : tcp_seq_foreign);
ack := *tcp_seq_flip_sense*(*ack_discard* : tcp_seq_local);
URG := *URG_discard*;
ACK := *ACK*; (* might be set in a bad SYN segment *)
PSH := *PSH_discard*;
RST := **F**; (* SYN segments never have *RST* set *)
SYN := *SYN*; (* might not be set in a bad segment to a listening socket *)
FIN := *FIN_discard*;
win := *win_discard*;
ws := *ws_discard*;
urp := *urp_discard*;
mss := *mss_discard*;
ts := *ts_discard*;
data := *data_discard*
⦈
) $\wedge$

(* Segment is addressed to an IP address belonging to one of the interfaces of host h and is not a link-layer multicast or IP-layer broadcast address *)
$i_1 \in$ local_ips *h.ifds* $\wedge$
$\neg$(is_broadormulticast *h.ifds* i_1)$\wedge$ (* very unlikely, since $i_1 \in$ local_ips *h.ifds* *)
$\neg$(is_broadormulticast *h.ifds* i_2) $\wedge$

(* Find the socket *sock* that has the best match for the address quad in segment *seg*, see tcp_socket_best_match (p86). Socket *sock* must have a form matching the patten Sock(...). *)
tcp_socket_best_match(*socks**sid*)(*sid*, *sock*)*seg* *h.arch* $\wedge$
sock = Sock($\uparrow$ *fid*, *sf*, is_1, $\uparrow p_1$, is_2, ps_2, *es*, *cantsndmore*, *cantrcvmore*,
TCP_Sock(LISTEN, *cb*, $\uparrow$ *lis*, *sndq*, *sndurp*, *rcvq*, *rcvurp*, *iobc*)) $\wedge$

(* If socket *sock* has a local IP address specified it should be the same as the destination IP address of segment *seg*. *)
(**case** is_1 **of** $\uparrow i1' \rightarrow i1' = i_1 \parallel * \rightarrow$ **T**) $\wedge$

(* A BSD socket in the LISTEN state may have its peer's IP address is_2 and port ps_2 set because listen() can be called from any TCP state. On other architectures they are both constrained to $*$. *)
$((is_2 = * \wedge ps_2 = *) \vee$
$(\text{bsd\_arch}\ \ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

(* Check that either: (a) the *SYN*, *ACK* flag combination is bad, or (b) the socket is illegally connecting to itself (Note: it is not possible to perform a self-connect once a socket is in the LISTEN state by using the sockets interface alone – it can only be achieved by a forged incoming segment. It is possible for a TCP socket to connect to itself but this is achieved through a sequence of socket calls that avoids entering the LISTEN state), or (c) the new incomplete connection can not be added to the incomplete connections queue because the queue of complete connections is full. *)
$(ACK \vee$
$(\neg SYN \wedge \neg ACK) \vee$
$(SYN \wedge \neg ACK \wedge i_1 = i_2 \wedge p_1 = p_2) \vee$
$\text{accept\_incoming\_q0}\ \ lis\ \mathbf{F}$
$) \wedge$

(* If an ACK with no SYN has been received send a RST segment, else just silently drop everything else. See dropwithreset (p120). *)
$(\mathbf{if}\ \neg SYN \wedge ACK\ \ \mathbf{then}$
$\quad \text{dropwithreset}\ \ seg\ h.ifds(\text{ticks\_of}\ \ h.ticks)\text{BANDLIM\_RST\_OPENPORT}\ bndlm\ bndlm'\ outsegs$
$\mathbf{else}$
$\quad outsegs = [\,] \wedge bndlm' = bndlm) \wedge$

(* Add the RST segment (if any) to the host's output queue, ignoring failure. See enqueue_and_ignore_fail (p118). *)
$\text{enqueue\_and\_ignore\_fail}\ \ h.arch\ h.rttab\ h.ifds\ outsegs\ oq\ oq'$

---

deliver_in_2 **tcp: network nonurgent** **Completion of active open (in SYN_SENT receive SYN,ACK and send ACK) or simultaneous open (in SYN_SENT receive SYN and send SYN,ACK)**

$$
\begin{array}{l}
h\ \langle\!|socks := socks \oplus \\
\quad [(sid, \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \\
\qquad\qquad cantsndmore, cantrcvmore, \text{TCP\_PROTO}\ tcp\_sock))]; \\
iq := iq; \\
oq := oq|\!\rangle \\
\xrightarrow{\tau}\ \ h\ \langle\!|socks := socks \oplus \\
\quad [(sid, \text{Sock}(\uparrow fid, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, \\
\qquad\qquad cantsndmore, cantrcvmore', \\
\qquad\qquad \text{TCP\_Sock}(st', cb'', *, [\,], *, rcvq', rcvurp', iobc')))]; \\
iq := iq'; \\
oq := oq'|\!\rangle
\end{array}
$$

$tcp\_sock = \text{TCP\_Sock0}(\text{SYN\_SENT}, cb, *, [\,], *, [\,], *, \text{NO\_OOBDATA}) \wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP}\ seg)) \wedge$

$(\exists win\_\ ws\_\ urp\_\ mss\_\ PSH\_discard.$
$win = \mathbf{w2n}\ \ win\_ \wedge$
$ws = \mathbf{option\_map}\ \mathbf{ord}\ \ ws\_ \wedge$
$urp = \mathbf{w2n}\ \ urp\_ \wedge$
$mss = \mathbf{option\_map}\ \ \mathbf{w2n}\ \ mss\_ \wedge$
$seg = \langle\!|$
$\qquad is_1 := \uparrow i_2;$
$\qquad is_2 := \uparrow i_1;$
$\qquad ps_1 := \uparrow p_2;$
$\qquad ps_2 := \uparrow p_1;$

$seq := tcp\_seq\_flip\_sense(seq : \mathrm{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack : \mathrm{tcp\_seq\_local});$
$URG := URG;$
$ACK := ACK;$
$PSH := PSH\_discard;$
$RST := \mathbf{F};$
$SYN := \mathbf{T};$
$FIN := FIN;$
$win := win\_;$
$ws := ws\_;$
$urp := urp\_;$
$mss := mss\_;$
$ts := ts;$
$data := data$
$\rrbracket) \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly *)

(* The ACK must be acceptable, else send RST. Typically (no data on active open), this is the same as $ack = iss + 1$ *)

$(ACK \implies (cb.iss < ack \wedge ack \leq cb.snd\_max)) \wedge$

(* resolve negotiated window scaling *)
$(\mathbf{case}\ (cb.request\_r\_scale, ws)\ \mathbf{of}$
$\quad (\uparrow rs, \uparrow ss) \rightarrow rcv\_scale' = rs \wedge$
$\qquad snd\_scale' = ss \wedge$
$\qquad tf\_doing\_ws' = \mathbf{T}\ \|$
$\quad \_15432 \rightarrow rcv\_scale' = 0 \wedge$
$\qquad snd\_scale' = 0 \wedge$
$\qquad tf\_doing\_ws' = \mathbf{F}) \wedge$

(* resolve negotiated timestamping *)
$tf\_rcvd\_tstmp' = \mathbf{is\_some}\ ts \wedge$
$tf\_doing\_tstmp' = (tf\_rcvd\_tstmp' \wedge cb.tf\_req\_tstmp) \wedge$

(* Note that for test generation at present we clear the route metric cache so this will always be NONE. BSD reads from the routing cache if there is an entry, otherwise passes NONE here. *)
$bw\_delay\_product\_for\_rt = * \wedge$

$\mathbf{let}\ ourmss = (\mathbf{case}\ cb.t\_advmss\ \mathbf{of}$
$\qquad * \rightarrow cb.t\_maxseg$ (* we did not advertise an MSS, so use the default value *)
$\quad \| \uparrow v \rightarrow v)\ \mathbf{in}$

$((rcvbufsize', sndbufsize', t\_maxseg'', snd\_cwnd') =$
$\quad \mathbf{if}\ mss \neq * \vee \neg\, \mathrm{bsd\_arch}\ h.arch\ \mathbf{then}$
$\qquad \mathrm{calculate\_buf\_sizes}\ ourmss\ mss\ bw\_delay\_product\_for\_rt$
$\qquad\quad (\mathrm{is\_localnet}\ h.ifds\ i_2)(sf.n(\mathrm{SO\_RCVBUF}))$
$\qquad\quad (sf.n(\mathrm{SO\_SNDBUF}))tf\_doing\_tstmp'\ h.arch$
$\quad \mathbf{else}$
$\qquad$ (* Note that since `tcp_mss()` is not called `snd_cwnd` remains at its initial (stupidly high) value. *)
$\qquad (sf.n(\mathrm{SO\_RCVBUF}), sf.n(\mathrm{SO\_SNDBUF}), cb.t\_maxseg, cb.snd\_cwnd)$
$) \wedge$

$sf' = sf\ \llparenthesis\ n := \mathrm{funupd\_list}\ sf.n[(\mathrm{SO\_RCVBUF}, rcvbufsize');$
$\qquad (\mathrm{SO\_SNDBUF}, sndbufsize')]\rrparenthesis \wedge$

$rcv\_window = \text{calculate\_bsd\_rcv\_wnd}\ \ sf'\ \ tcp\_sock \wedge$

$\mathbf{let}\ (t\_softerror', t\_rttseg', t\_rttinf', tt\_rexmt')$
$= (\mathbf{if}\ \ ACK\ \ \mathbf{then}$
(* completion of active open. Conditions originally copied verbatim from *deliver_in_3*. *)

(* update RTT estimators from timestamp or roundtrip time *)
$\mathbf{let}\ \ emission\_time = \mathbf{case}\ \ ts\ \ \mathbf{of}$
$\uparrow(ts\_val, ts\_ecr) \rightarrow \uparrow(ts\_ecr - 1)$
$\| * \rightarrow$
$(\mathbf{case}\ \ cb.t\_rttseg\ \ \mathbf{of}$
$\uparrow(ts_0, seq_0) \rightarrow \mathbf{if}\ \ ack > seq_0$
$\mathbf{then}\ \uparrow ts_0$
$\mathbf{else}\ \ *$
$\| * \rightarrow *)\ \mathbf{in}$

(* clear soft error, cancel timer, and update estimators if we successfully timed a segment round-trip *)
$\mathbf{let}\ (t\_softerror', t\_rttseg', t\_rttinf')$
$= \mathbf{if}\ \ \mathbf{is\_some}\ \ emission\_time\ \ \mathbf{then}$
$(*,$
$*,$
$\text{update\_rtt}(\text{real\_of\_int}(\text{ticks\_of}\ \ h.ticks - \mathbf{the}\ \ emission\_time)/\,\text{HZ})$
$cb.t\_rttinf)$
$\mathbf{else}$
$(cb.t\_softerror,$
$cb.t\_rttseg,$
$cb.t\_rttinf)\ \mathbf{in}$

(* mess with retransmit timer if appropriate *)
$\mathbf{let}\ \ tt\_rexmt' =$
$(\mathbf{if}\ \ ack = cb.snd\_max\ \ \mathbf{then}$
(* if acked everything, stop *)
$*$
(* $needoutput = 1$ – see below *)
$\mathbf{else}\ \ \mathbf{if}\ \ \text{mode\_of}\ \ cb.tt\_rexmt = \uparrow \text{RexmtSyn}\ \ \mathbf{then}$
(* if partial ack, restart from current backoff value, which is always zero because of the above updates to the RTT estimators and shift value. *)
$\text{start\_tt\_rexmtsyn}\ \ h.arch\ 0\ \mathbf{T}\ \ t\_rttinf'$
$\mathbf{else}\ \ \mathbf{if}\ \ \text{mode\_of}\ \ cb.tt\_rexmt\ \in \{*; \uparrow \text{Rexmt}\}\ \mathbf{then}$
(* ditto *)
$\text{start\_tt\_rexmt}\ \ h.arch\ 0\ \mathbf{T}\ \ t\_rttinf'$
$\mathbf{else}\ \ \mathbf{if}\ \ emission\_time \neq *\ \ \mathbf{then}$
$\mathbf{case}\ \ cb.tt\_rexmt\ \ \mathbf{of}$
(* bizarre but true. `tcp_input.c:1766` says c.f. Phil Karn's retransmit algorithm *)
$* \rightarrow *$
$\| \uparrow(((mode, shift))_d) \rightarrow \uparrow(((mode, 0))_d)$
$\mathbf{else}$
(* do nothing *)
$cb.tt\_rexmt$
$)\ \mathbf{in}$
$(t\_softerror',$
$t\_rttseg',$
$t\_rttinf',$
$tt\_rexmt')$
$\mathbf{else}$
(* simultaneous open *)
$(cb.t\_softerror,$

$cb.t\_rttseg$,
$cb.t\_rttinf$,
start_tt_rexmt $h.arch$ 0 **T** $cb.t\_rttinf$) (* reset rexmt timer *)
) **in**

(* urgent pointer processing. See *deliver_in_3* for discussion (these conditions are originally copied verbatim from there). *)
$(\exists iobc\ rcvurp.$
$iobc = \mathrm{NO\_OOBDATA} \wedge$ (* we know the initial state has no OOB data *)
$rcvurp = * \wedge$
(**if** $URG \wedge$
$urp > 0 \wedge$

$urp + 0 \leq \mathrm{SB\_MAX}$
then
(**if** $seq + urp > cb.rcv\_up$ **then**
$rcv\_up' = seq + 1 + urp \wedge$

$rcvurp' = \uparrow(0 + \mathbf{num}(seq + urp - cb.rcv\_nxt))$
else
$rcv\_up' = cb.rcv\_nxt \wedge$ (* pull along with window *)
$rcvurp' = rcvurp) \wedge$
(**if** $urp \leq \mathbf{length}\ data \wedge sf.b(\mathrm{SO\_OOBINLINE}) = \mathbf{F}$ **then**
$iobc' = \mathrm{OOBDATA}(EL(urp - 1) data) \wedge$
$data\_deoobed = (\mathrm{TAKE}(urp - 1) data) \mathbin{@} (\mathrm{DROP}\ urp\ data)$
else
$iobc' = ($**if** $seq + urp > cb.rcv\_up$ **then** NO_OOBDATA **else** $iobc) \wedge$
$data\_deoobed = data)$
else
$rcv\_up' = seq + 1 \wedge$
$rcvurp' = rcvurp \wedge$
$iobc' = iobc \wedge$
$data\_deoobed = data)$
$) \wedge$

(* 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). *)
$data' = \mathrm{TAKE}\ rcv\_window\ data\_deoobed \wedge$
$FIN' = ($**if** $data' = data\_deoobed$ **then** FIN **else** $\mathbf{F}) \wedge$
$rcvq' = data' \wedge$ (* because $rcvq$ is empty initially *)
$rcv\_nxt' = seq + 1 + \mathbf{length}\ data' + ($**if** FIN' **then** 1 **else** $0) \wedge$
$rcv\_wnd' = rcv\_window - \mathbf{length}\ data' \wedge$

$cb' = cb\ \llparenthesis$
$tt\_rexmt := tt\_rexmt'$;
(* not persist, because we do not have any data to send *)
$t\_idletime :=$ stopwatch_zero; (* just received a segment *)
$tt\_keep := \uparrow((())_{\mathrm{slow\_timer\ TCPTV\_KEEP\_IDLE}})$;
$tt\_conn\_est := *$;
$tt\_delack := *$;

$snd\_una \mathrel{:\hat{=}} ack$ **onlywhen** ACK; (* = cb.iss + 1, or +2 if full ack of SYN,FIN *)
$snd\_nxt \mathrel{:\hat{=}} ack$ **onlywhen**$(ACK \wedge cantsndmore)$; (* prepare for possible outbound FIN *)
$snd\_max \mathrel{:\hat{=}} ack$ **onlywhen**$(ACK \wedge cantsndmore \wedge ack > cb.snd\_max)$;
(* we doubt snd_max can ever increase here, but put this in for safety *)

$snd\_wl1 := \mathbf{if}\ ACK\ \mathbf{then}\ seq + 1\ \mathbf{else}\ seq$; (* must update window. c.f. TCPv2p951, TCPv2p981f, and `tcp_input.c:1824` *)
$snd\_wl2\ :\hat{=}\ ack\ \mathbf{onlywhen}\ ACK$;
$snd\_wnd := win \ll snd\_scale'$;
$snd\_cwnd := \mathbf{if}\ ACK \wedge ack > cb.iss + 1\ \mathbf{then}$
(* BSD clamps `snd_cwnd` to the maximum window size (65535), but only if we received an ack for data other than the initial SYN. See `tcp_input.c::1791` *)
$\mathbf{min}(snd\_cwnd')(\mathrm{TCP\_MAXWIN} \ll snd\_scale')$
else
snd_cwnd';
$rcv\_scale := rcv\_scale'$;
$snd\_scale := snd\_scale'$;
$tf\_doing\_ws := tf\_doing\_ws'$;
$irs := seq$;
$rcv\_nxt := rcv\_nxt'$;
$rcv\_wnd := rcv\_wnd'$;
$tf\_rxwin0sent :=(rcv\_wnd' = 0)$;
$rcv\_adv := rcv\_nxt' + (rcv\_wnd' \gg rcv\_scale') \ll rcv\_scale'$;
$rcv\_up := rcv\_up'$;
$t\_maxseg := t\_maxseg''$;
$ts\_recent := \mathbf{case}\ ts\ \mathbf{of}$
(* record irrespective of whether we negotiated to do this or not, like BSD *)
$* \rightarrow cb.ts\_recent\ \|$
$\uparrow(ts\_val, ts\_ecr) \rightarrow (ts\_val)^{\mathrm{TimeWindow}}_{\mathrm{kern\_timer\ \ dtsinval}}$;
(* timestamp will become invalid in 24 days *)
$last\_ack\_sent := rcv\_nxt'$;
$t\_softerror := t\_softerror'$;
$t\_rttseg := t\_rttseg'$;
$t\_rttinf := t\_rttinf'$;
$tf\_req\_tstmp := tf\_doing\_tstmp'$;
$tf\_doing\_tstmp := tf\_doing\_tstmp'$
$\rrparenthesis \wedge$

(* now generate seg', unless we're delaying the ACK *)
$(\mathbf{choose}\ seg' :: (\mathbf{if}\ ACK\ \mathbf{then}$
(* completion of active open *)
$\mathrm{make\_ack\_segment}\ cb'(cantsndmore \wedge ack < cb.iss + 2)(i_1, i_2, p_1, p_2)(\mathrm{ticks\_of}\ h.ticks)$
else
(* simultaneous open *)

$\mathbf{let}\ cb''' =$
$(\mathbf{if}\ ((\mathrm{linux\_arch}\ h.arch) \wedge cb.tf\_req\_tstmp)\ \mathbf{then}$
$cb'\ \llparenthesis\ tf\_req\_tstmp := \mathbf{T};$
$tf\_doing\_tstmp := \mathbf{T} \rrparenthesis$
else
$cb')\ \mathbf{in}$

$(\mathbf{if}\ \mathrm{bsd\_arch}\ h.arch\ \mathbf{then}$
$\mathrm{make\_ack\_segment}\ cb'''\ \mathbf{F}(i_1, i_2, p_1, p_2)(\mathrm{ticks\_of}\ h.ticks)$
else
$\mathrm{make\_syn\_ack\_segment}\ cb'''(i_1, i_2, p_1, p_2)(\mathrm{ticks\_of}\ h.ticks)))$.

(* Add the segment to the host's output queue. See enqueue_or_fail (p118). *)

enqueue_or_fail $\mathbf{T}$ $h.arch$ $h.rttab$ $h.ifds$[TCP seg']oq
$(cb \ \langle\!\langle \ t\_rttinf := cb'.t\_rttinf;$
$t\_maxseg := t\_maxseg'';$
$snd\_nxt := cb.snd\_nxt;$
$tt\_delack := cb.tt\_delack;$
$last\_ack\_sent := cb.last\_ack\_sent;$
$rcv\_adv := cb.rcv\_adv$
$\rangle\!\rangle) cb'(cb'', oq')$
$) \wedge$

(* Note that we change state even if enqueuing or routing returned an error, trusting to retransmit to solve our problem. *)

(**if** ACK **then**
(* completion of active open *)
(**if** $\neg FIN'$ **then**
$(cantrcvmore' = cantrcvmore \wedge$
$st' =$
(**if** $cantsndmore = \mathbf{F}$ **then**
ESTABLISHED
else if $cb.snd\_max > cb.iss + 1 \wedge ack \geq cb.snd\_max$ **then** (* our FIN is ACKed *)
FIN_WAIT_2
else
FIN_WAIT_1)) (* we were trying to send a FIN from SYN_SENT, so move straight to FIN_WAIT_2. Definitely the case with BSD; should also be true for other archs. *)
else
$(cantrcvmore' = \mathbf{T} \wedge$
$st' =$
(**if** $cantsndmore = \mathbf{F}$ **then**
CLOSE_WAIT
else
LAST_ACK))) (* we were trying to send a FIN from SYN_SENT and also receive a FIN, so we move straight into LAST_ACK. *)
else
(* simultaneous open *)
(**if** $\neg FIN'$ **then**
$(st' =$ SYN_RECEIVED $\wedge$
$cantrcvmore' = cantrcvmore)$
else

$(st' =$ CLOSE_WAIT$\wedge$ (* yes, really! (in BSD) even though we've not yet had our initial SYN acknowledged! See `tcp_input.c:2065 +/-2000` *)

$cantrcvmore' = \mathbf{T}))$
)

---

deliver_in_2a **tcp: network nonurgent** **Receive bad or boring datagram and RST or ignore for SYN_SENT socket**

$$h \ \langle\!\langle socks := socks \oplus [(sid, sock)]; iq := iq; oq := oq; bndlm := bndlm \rangle\!\rangle \xrightarrow{\tau} h \ \langle\!\langle socks := socks \oplus [(sid, sock')]; iq := iq'; oq := oq'; bndlm := bndlm' \rangle\!\rangle$$

(* **Summary:** For a SYN_SENT socket unacceptable acks get RSTed; boring but otherwise OK segments are ignored. *)

$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(\text{SYN\_SENT}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$

$(\exists seq\_discard\ URG\_discard\ PSH\_discard\ FIN\_discard$
$\quad win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard.$
$seg = \langle\!|$
$\quad is_1 := \uparrow i_2;$
$\quad is_2 := \uparrow i_1;$
$\quad ps_1 := \uparrow p_2;$
$\quad ps_2 := \uparrow p_1;$
$\quad seq := tcp\_seq\_flip\_sense(seq\_discard : \text{tcp\_seq\_foreign});$
$\quad ack := tcp\_seq\_flip\_sense(ack : \text{tcp\_seq\_local});$
$\quad URG := URG\_discard;$
$\quad ACK := ACK;$
$\quad PSH := PSH\_discard;$
$\quad RST := \mathbf{F};$
$\quad SYN := SYN;$
$\quad FIN := FIN\_discard;$
$\quad win := win\_discard;$
$\quad ws := ws\_discard;$
$\quad urp := urp\_discard;$
$\quad mss := mss\_discard;$
$\quad ts := ts\_discard;$
$\quad data := data\_discard$
$\quad |\!\rangle$
$) \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

$((ACK \wedge \neg(cb.iss < ack \wedge ack \leq cb.snd\_max)) \vee$
$(\neg SYN \wedge (\neg ACK \vee (ACK \wedge cb.iss < ack \wedge ack \leq cb.snd\_max)))) \wedge$

$(\mathbf{if}\ ACK \wedge \neg(cb.iss < ack \wedge ack \leq cb.snd\_max)\ \mathbf{then}$
$\quad \text{dropwithreset}\ seg\ h.ifds(\text{ticks\_of}\ h.ticks)\text{BANDLIM\_UNLIMITED}\ bndlm\ bndlm'\ outsegs$
$\mathbf{else\ if}\ \neg SYN \wedge (\neg ACK \vee (ACK \wedge cb.iss < ack \wedge ack \leq cb.snd\_max))\ \mathbf{then}$
$\quad outsegs = [\,] \wedge bndlm' = bndlm$
$\mathbf{else}$
$\quad \mathbf{F}) \wedge$
$\mathbf{let}\ tcp\_sock = \text{tcp\_sock\_of}\ sock\ \mathbf{in}$
(* BSD `rcv_wnd` bug: the receive window updated code in `tcp_input` gets executed *before* the segment is processed, so even for bad segments, it gets updated. *)
$\mathbf{let}\ rcv\_window = \text{calculate\_bsd\_rcv\_wnd}\ sf\ tcp\_sock\ \mathbf{in}$
$sock' = sock\ \langle\!|\ pr := \text{TCP\_PROTO}(tcp\_sock$
$\quad \langle\!|\ cb := tcp\_sock.cb$
$\quad\quad \langle\!|\ rcv\_wnd := \mathbf{if}\ \text{bsd\_arch}\ h.arch\ \mathbf{then}\ rcv\_window\ \mathbf{else}\ tcp\_sock.cb.rcv\_wnd;$
$\quad\quad rcv\_adv := \mathbf{if}\ \text{bsd\_arch}\ h.arch\ \mathbf{then}\ tcp\_sock.cb.rcv\_nxt + rcv\_window$
$\quad\quad\quad \mathbf{else}\ tcp\_sock.cb.rcv\_adv;$
$\quad\quad t\_idletime := \text{stopwatch\_zero};$
$\quad\quad tt\_keep := \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_IDLE}})$
$\quad\quad |\!\rangle |\!\rangle)|\!\rangle \wedge$
$\text{enqueue\_and\_ignore\_fail}\ h.arch\ h.rttab\ h.ifds\ outsegs\ oq\ oq'$

deliver_in_3 **tcp: network nonurgent** **Receive data, FINs, and ACKs in a connected state**

$h\ \langle\!|socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $h\ \langle\!|socks := socks';$
$iq := iq;$ $\quad iq := iq';$
$oq := oq;$ $\quad oq := oq';$
$bndlm := bndlm|\!\rangle$ $\quad bndlm := bndlm'|\!\rangle$

$sid\ \notin (\mathbf{dom}(socks)) \wedge$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$

(* Assert that the socket meets some sanity properties. This is logically superfluous but aids semi-automatic model checking. See sane_socket (p84) for further details. *)
sane_socket $sock \wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
dequeue_iq$(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

(* 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. *)
$(\exists win\_\ urp\_\ ws\_discard\ mss\_discard.$
$seg = \langle\!|$
$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq : \mathrm{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack : \mathrm{tcp\_seq\_local});$
$URG := URG;$ (* Urgent/OOB data is processed by this rule *)
$ACK := ACK;$ (* Acknowledgements are processed *)
$PSH := PSH;$ (* Push flag maybe set on an incoming data segment *)
$RST := \mathbf{F};$ (* *RST* segments are not handled by this rule *)
$SYN := SYN;$ (* *SYN* flag set may be set in the final segment of a simultaneous open *)
$FIN := FIN;$ (* Processing of FIN flag handled *)
$win := win\_;$
$ws := ws\_discard;$
$urp := urp\_;$
$mss := mss\_discard;$
$ts := ts;$
$data := data$ (* Segment may have data *)
$|\!\rangle \wedge$

(* Equality of some type casts, and application of the socket's send window scaling to the received window advertisment *)
$win = \mathbf{w2n}\ win\_ \ll tcp\_sock.cb.snd\_scale \wedge$
$urp = \mathbf{w2n}\ urp\_$
$) \wedge$

(* 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. *)
$tcp\_sock.st\ \notin \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\} \wedge$
$tcp\_sock.st\ \in \{\mathrm{SYN\_RECEIVED}; \mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}; \mathrm{FIN\_WAIT\_1}; \mathrm{FIN\_WAIT\_2};$
$\quad \mathrm{CLOSING}; \mathrm{LAST\_ACK}; \mathrm{TIME\_WAIT}\} \wedge$

(* For a socket in the SYN_RECEIVED state check that the ACK is valid (the acknowledge value *ack* is not outside the range of sequence numbers that have been transmitted to the remote socket) and that the segment is not a LAND DoS attack (the segment's sequence number is not smaller than the remote socket's (the receiver from this socket's perspective) initial sequence number) *)
$\neg(tcp\_sock.st = \text{SYN\_RECEIVED} \wedge$
$((ACK \wedge (ack \leq tcp\_sock.cb.snd\_una \vee ack > tcp\_sock.cb.snd\_max)) \vee$
$seq < tcp\_sock.cb.irs)) \wedge$

(* If socket *sock* has previously emitted a *FIN* segment check that a thread is still associated with the socket, i.e. check that the socket still has a valid file identifier $fid \neq *$. If not, and the segment contains new data, the segment should not be processed by this rule as there is no thread to read the data from the socket after processing. Query: how does this *st* condition relate to *wesentafin* below? *)
$\neg(tcp\_sock.st \in \{\text{FIN\_WAIT\_1}; \text{CLOSING}; \text{LAST\_ACK}; \text{FIN\_WAIT\_2}; \text{TIME\_WAIT}\} \wedge$
$sock.fid = * \wedge$
$seq + \textbf{length}\ data > tcp\_sock.cb.rcv\_nxt) \wedge$

(* A *SYN* should be received only in the SYN_RECEIVED state. *)
$(SYN \implies tcp\_sock.st = \text{SYN\_RECEIVED}) \wedge$

(* Socket *sock* has previously sent a *FIN* segment iff *snd_max* is strictly greater than the sequence number of the byte after the last byte in the send queue *sndq*. *)
let $wesentafin = tcp\_sock.cb.snd\_max > tcp\_sock.cb.snd\_una + \textbf{length}\ tcp\_sock.sndq$ **in**

(* If the socket *sock* has previously sent a *FIN* segment it has been acknowledged by segment *seg* if the segment has the *ACK* flag set and an acknowledgment number $ack \geq cb.snd\_max$. *)
let $ourfinisacked = (wesentafin \wedge ACK \wedge ack \geq tcp\_sock.cb.snd\_max)$ **in**

(* Process the segment and return an updated socket state *)
(* The segment processing is performed by the four relations below, i.e., di3_topstuff, di3_ackstuff, di3_datastuff and di3_ststuff. Each of these relates a socket and bandwidth limiter state before the segment is processed to a tuple containing an updated socket, new bandwidth limiter state, a list of zero or more segments to output and a continue flag. The aim is to model the progression of the segment through `tcp_input()`. When the continue flag is **T** segment processing should continue. The infix function andThen applies the function on its left hand side and only continues with the function on its right hand side if the left hand function's continue flag is **T**. For a further explanation of this relational monad behaviour see *aux_relmonad* (p**??**). *)
let *topstuff* =
(* Initial processing of the segment: PAWS (protection against wrap sequence numbers); ensure segment is not entirely off the right hand edge of the window; timer updates, etc. For further information see di3_topstuff (p294).*)
di3_topstuff *seg h.arch h.rttab h.ifds*(ticks_of *h.ticks*)
and *ackstuff* =
(* Process the segment's acknowledgement number and do congestion control. See di3_ackstuff (p298).*)
di3_ackstuff *tcp_sock seg ourfinisacked h.arch h.rttab h.ifds*(ticks_of *h.ticks*)
and *datastuff theststuff* =
(* Extract and reassemble data (including urgent data). See di3_datastuff (p304). *)
di3_datastuff *theststuff tcp_sock seg ourfinisacked h.arch*
and *ststuff FIN_reass* =
(* Possibly change the socket's state (especially on receipt of a valid *FIN*). See di3_ststuff (p305). *)
di3_ststuff *FIN_reass ourfinisacked ack*
in
(*topstuff* andThen
ackstuff andThen
datastuff ststuff)
(*sock*, *bndlm*) (* state before *)
$((sock', bndlm', outsegs), continue') \wedge$ (* state after *)

$sock'.pr = \text{TCP\_PROTO}(tcp\_sock') \wedge$

(* If socket *sock* was initially in the SYN_RECEIVED state and after processing *seg* is in the ESTABLISHED state (or if the segment contained a *FIN* and the socket is in one of the FIN_WAIT_1, FIN_WAIT_2 or CLOSE_WAIT states), the socket is probably on some other socket's incomplete connections queue and *seg* is the final segment in a passive open. If it is on some other socket's incomplete connections queue the other socket is updated to move the newly connected socket's reference from the incomplete to the complete connections queue (unless the complete connection queue is full, in which case the new connection is dropped and all references to it are removed). If not, *seg* is the final segment in a simultaneous open in which case no other sockets are updated. The auxiliary function di3_socks_update (p308) does all the hard work, updating the relevant sockets in the finite map *socks* to yield *socks'*. *)

$(\textbf{if } \mathit{tcp\_sock.st} = \text{SYN\_RECEIVED} \wedge$
$\quad \mathit{tcp\_sock'.st} \in \{\text{ESTABLISHED}; \text{FIN\_WAIT\_1}; \text{FIN\_WAIT\_2}; \text{CLOSE\_WAIT}\}$ **then**
$\quad\quad \text{di3\_socks\_update } \mathit{sid}(\mathit{socks} \oplus (\mathit{sid}, \mathit{sock'}))\mathit{socks'}$
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. *)
$\quad \mathit{socks'} = \mathit{socks} \oplus (\mathit{sid}, \mathit{sock'})) \wedge$

(* Queue any segments for output on the host's output queue. In the common case there are no segments to be output as output is handled by *deliver_out_1* etc. The exception is that di3_ackstuff (and its auxiliaries) require an immediate *ACK* segment to be emitted under certain congestion control conditions. See di3_ackstuff (p298) and di3_newackstuff (p295) for further details. *)
$\text{enqueue\_oq\_list\_qinfo}(\mathit{oq}, \mathit{outsegs}, \mathit{oq'})$

– *deliver_in_3* **initial checks :**

$\text{di3\_topstuff } \mathit{seg}\ \mathit{arch}\ \mathit{rttab}\ \mathit{ifds}\ \mathit{ticks} =$
(* monadic state accessor: *sock* is the socket processing the segment, as determined by *deliver_in_3* *)
$(\text{get\_sock } \lambda \mathit{sock}.$
(* Pull out the TCP protocol and control blocks *)
$\textbf{let } \mathit{tcp\_sock} = \text{tcp\_sock\_of } \mathit{sock} \textbf{ in}$
$\textbf{let } \mathit{cb} = \mathit{tcp\_sock.cb} \textbf{ in}$

(* If the segment has the *SYN* flag set, increment the sequence number so that it is the sequence number of the first byte of data in the segment *)
$\textbf{let } \mathit{seq} = \mathit{tcp\_seq\_flip\_sense}\ \mathit{seg.seq} + (\textbf{if } \mathit{seg.SYN} \textbf{ then } 1 \textbf{ else } 0) \textbf{ in}$
(* The sequence number of the byte logically after the last byte of data in the segment *)
$\textbf{let } \mathit{rseq} = \mathit{seq} + \textbf{length } \mathit{seg.data} \textbf{ in}$
$\textbf{let } \mathit{ts} = \mathit{seg.ts} \textbf{ in}$

(* PAWS (Protection Against Wrapped Sequence numbers) check: If the segment contains a timestamp value that is strictly less than *ts_recent* then the segment is invalid and the PAWS check fails. The value *ts_recent* is the timestamp value of the most recent of the previous segments that was successfully processed, i.e., the last segment that *deliver_in_3* processed without dropping. *)
$\textbf{let } \mathit{paws\_failed} =$
$(\exists \mathit{ts\_val}\ \mathit{ts\_ecr}\ \mathit{ts\_recent}.$
$\quad \mathit{ts} = \uparrow(\mathit{ts\_val}, \mathit{ts\_ecr}) \wedge$ (* segment's timestamp field is a pair *)
$\quad \text{timewindow\_val\_of } \mathit{cb.ts\_recent} = \uparrow \mathit{ts\_recent} \wedge$ (* most recent timestamp recorded *)
$\quad \mathit{ts\_val} < \mathit{ts\_recent}) \textbf{ in}$ (* check the segment's timestamp is not old *)

(* If the segment lies entirely off the right-hand edge of *sock*'s receive window then it should be dropped, provided it is not a window probe. *)
$\textbf{let } \mathit{segment\_off\_right\_hand\_edge} =$
$(\textbf{let } \mathit{rcv\_wnd'} = \text{calculate\_bsd\_rcv\_wnd } \mathit{sock.sf}\ \mathit{tcp\_sock} \textbf{ in}$ (* size of receive window *)
$\quad (\mathit{seq} \geq \mathit{cb.rcv\_nxt} + \mathit{rcv\_wnd'}) \wedge$ (* segment starts on or after the right hand edge *)
$\quad (\mathit{rseq} > \mathit{cb.rcv\_nxt} + \mathit{rcv\_wnd'}) \wedge$ (* segment ends after the right hand edge *)
$\quad (\mathit{rcv\_wnd'} \neq 0)) \textbf{ in}$ (* The segment is not a window probe, i.e., *rcv_wnd'* is not zero *)

(* Drop the segment being processed if either the PAWS check or the "off right hand edge of window" checks fail *)
$\textbf{let } \mathit{drop\_it} = (\mathit{paws\_failed} \vee \mathit{segment\_off\_right\_hand\_edge}) \textbf{ in}$

(* The value *ts_recent* will be updated to hold the value of the segment's timestamp field if the segment is not dropped. Timestamps are invalidated after 24 days - this is ensured by the attached kernel timer kern_timer dtsinval. *)
let $ts\_recent' = (\mathbf{fst}(\mathbf{the}\ ts))^{\text{TimeWindow}}_{\text{kern\_timer dtsinval}}$ **in**

(* Reset the socket's idle timer and keepalive timer to start counting from zero as activity is taking place on the socket: a segment is being processed. If the FIN_WAIT_2 timer is enabled this may be reset upon processing this segment. See update_idle (p119) for further details *)
let $(t\_idletime', tt\_keep', tt\_fin\_wait\_2')$ = update_idle *tcp_sock* **in**

(* Using the monadic state accessor modify_cb (p**??**), update the socket's control block with the new timer values and the most recent timestamp seen.
The *ts_recent* field is only updated if the segment currently being processed is not scheduled to be dropped, has a timestamp value set and is from a segment whose first byte of data has sequence number less than or equal to the last acknowledgement number sent in a segment to the remote end. The last condition (when coupled with the PAWS check above) ensures that *ts_recent* only increases monotonically and as is only updated by either a duplicate segment with a newer timestamp, or the next in-order segment expected by the receiving socket with a newer timestamp. It would be incorrect to record the newer timestamps of out-of-order segments because they would fail the PAWS check and get dropped
Note: if a reasonably continuous stream of segments is being received with increasing timestamp values and few data segments are sent in return such that acknowledgments are delayed, i.e., every other segment is acknowledged), then only the timestamp from every other segment is recorded by these conditions. This is still sufficient to protect against wrapped sequence numbers. *)
modify_cb($\lambda cb'.cb'$ ⦃ $tt\_keep := tt\_keep'$;
$tt\_fin\_wait\_2 := tt\_fin\_wait\_2'$;
$t\_idletime := t\_idletime'$;
$ts\_recent \mathrel{:\hat{=}} ts\_recent'$ **onlywhen**
$(\neg drop\_it \wedge$ **is_some** $ts \wedge seq \leq cb.last\_ack\_sent)$
⦄) andThen

if *drop_it* **then**
(* Decided to drop the segment. mlift_dropafterack_or_fail (p120) may decide to *RST* the connection depending upon the socket state. If so, the *RST* segment is retained on the monadic output segment list returned to *deliver_in_3* for queueing. *)
mlift_dropafterack_or_fail *seg arch rttab ifds ticks* andThen
(* After dropping, stop processing the segment. No need to waste time processing the segment any further *)
stop
else
(* Otherwise the segment is valid so allow processsing to continue. *)
cont
)

– *deliver_in_3* **new ack processing, used in** di3_ackstuff **:**
di3_newackstuff *tcp_sock_0 seg ourfinisacked arch rttab ifds ticks* =
(* Pull some fields out of the segment *)
let $ack = tcp\_seq\_flip\_sense\ seg.ack$ **in**
let $ts = seg.ts$ **in**

(* Get the socket's control block using the monadic state accessor get_cb. *)
(get_cb $\lambda cb'$.

(**if** $\neg$ TCP_DO_NEWRENO $\vee cb'.t\_dupacks < 3$ **then**
(* If not doing NewReno-style Fast Retransmit or there have been fewer than 3 duplicate *ACKS* then clear the duplicate *ACK* counter. If there were more than 3 duplicate *ACKS* previously then the congestion window was inflated as per RFC2581 so retract it to *snd_ssthresh* *)
modify_cb($\lambda cb'.cb'$ ⦃ $t\_dupacks := 0$;
$snd\_cwnd \mathrel{:\hat{=}} (\mathbf{min}\ cb'.snd\_cwnd\ cb'.snd\_ssthresh)$ (* retract the window safely *)

onlywhen$(cb'.t\_dupacks \geq 3)\rrparenthesis)$

else if TCP_DO_NEWRENO $\wedge cb'.t\_dupacks \geq 3 \wedge ack < cb'.snd\_recover$ **then**
(* The host supports NewReno-style Fast Recovery, the socket has received at least three duplicate *ACK*s previously and the new *ACK* does not complete the recovery process, i.e., there are further losses or network delays. The new *ACK* is a partial *ACK* per RFC2582. Perform a retransmit of the next unacknowledged segment and deflate the congestion window as per the RFC. *)
modify_cb($\lambda cb'.cb' \llparenthesis$
(* Clear the retransmit timer and round-trip time measurement timer. These will be started by tcp_output_really when the retransmit is actioned. *)
$tt\_rexmt := *$;
$t\_rttseg := *$;

(* Segment to retransmit starts here *)
$snd\_nxt := ack$;

(* Allow one segment to be emitted *)
$snd\_cwnd := cb'.t\_maxseg$
$\rrparenthesis$) andThen

(* Attempt to create a segment for output using the modified control block (this is a relational monad idiom) *)
mlift_tcp_output_perhaps_or_fail *ticks arch rttab ifds* andThen

(* Finally update the control block: *)
modify_cb($\lambda cb'.cb' \llparenthesis$
(* RFC2582 partial window deflation: deflate the congestion window by the amount of data freshly acknowledged and add back one maximum segment size *)
$snd\_cwnd :=$ **num**(**int_of_num** $cb'.snd\_cwnd -$
$(ack - cb'.snd\_una) +$ **int_of_num** $cb'.t\_maxseg)$;
$snd\_nxt := cb'.snd\_nxt \rrparenthesis$) (* restore previous value *)

else if TCP_DO_NEWRENO $\wedge cb'.t\_dupacks \geq 3 \wedge ack \geq cb'.snd\_recover$ **then**
(* The host supports NewReno-style Fast Recovery, the socket has received at least three duplicate *ACK* segments and the new *ACK* acknowledges at least everything upto *snd_recover*, completing the recovery process. *)

modify_cb($\lambda cb'.cb' \llparenthesis$ $t\_dupacks := 0$; (* clear the duplicate *ACK* counter *)
(* Open up the congestion window, being careful to avoid an RFC2582 Ch3.5 Pg6 "burst of data". *)
$snd\_cwnd :=($
if $cb'.snd\_max - ack <$ **int_of_num** $cb'.snd\_ssthresh$ **then**
(* If *snd_ssthresh* is greater than the number of bytes of data still unacknowledged and presumed to be in-flight, set *snd_cwnd* to be one segment larger than the total size of all the segments in flight. This is burst avoidance: *tcp_output* is only able to send upto one further segment until some of the in flight data is acknowledged. *)
num$(cb'.snd\_max - ack +$ **int_of_num** $cb'.t\_maxseg)$
else
(* Otherwise, set *snd_cwnd* to be *snd_ssthresh*, forbidding any further segment output until some in flight data is acknowledged.*)
$cb'.snd\_ssthresh)$
$\rrparenthesis$)

else *assert_failure*"di3_newackstuff" (* impossible *)

) andThen

(* Check *ack* value is sensible, i.e., not greater than the highest sequence number transmitted so far *)
if $ack > cb'.snd\_max$ **then**
(* Drop the segment and possibly emit a *RST* segment *)
mlift_dropafterack_or_fail *seg arch rttab ifds ticks* andThen
stop

else (* continue processing *)
(* If the retransmit timer is set and the socket has done only one retransmit and it is still within the bad retransmit timer window, then because this is an *ACK* of new data the retransmission was done in error. Flag this so that the control block can be recovered from retransmission mode. This is known as a "bad retransmit". *)
let $revert\_rexmt = (\text{mode\_of } cb'.tt\_rexmt \in \{\uparrow \text{REXMT}; \uparrow \text{REXMTSYN}\} \wedge$
$\text{shift\_of } cb'.tt\_rexmt = 1 \wedge \text{timewindow\_open } cb'.t\_badrxtwin)$ **in**

(* Attempt to calculate a new round-trip time estimate *)
let $emission\_time =$ **case** $(ts, cb'.t\_rttseg)$ **of**
$(\uparrow(ts\_val, ts\_ecr), \_) \rightarrow$
(* By using the segment's timestamp if it has one *)
$\uparrow(ts\_ecr - 1)$
$\| (*, \uparrow(ts_0, seq_0)) \rightarrow$
(* Or if not, by the control blocks round-trip timer, if it covers the segment(s) being acknowledged *)
if $ack > seq_0$ **then** $\uparrow ts_0$ **else** $*$
$\| (*, *) \rightarrow$
(* Otherwise, it is not possible to calculate a round-trip update *)
$*$ **in**

(* If a new round-trip time estimate was calculated above, update the round-trip information held by the socket's control block *)
let $t\_rttinf' =$ **case** $emission\_time$ **of**
$\uparrow t\_rttinf \rightarrow \text{update\_rtt}(\text{real\_of\_int}(ticks - \textbf{the } emission\_time)/\text{HZ})$
$cb'.t\_rttinf$
$\| * \rightarrow cb'.t\_rttinf$ **in**

(* Update the retransmit timer *)
let $tt\_rexmt' =$
(**if** $ack = cb'.snd\_max$ **then**
$*$ (* If all sent data has been acknowledged, disable the timer *)
else case mode_of $cb'.tt\_rexmt$ **of**
$* \rightarrow$
(* If not set, set it as there is still unacknowledged data *)
start_tt_rexmt *arch* 0 **T** t_rttinf'
$\| \uparrow$ REXMT $\rightarrow$
(* If set, reset it as a new acknowledgement segment has arrived *)
start_tt_rexmt *arch* 0 **T** t_rttinf'
$\| \_444 \rightarrow$
(* Otherwise, leave it alone. The timer will never be in REXMTSYN here and the only other case is PERSIST, in which case it should be left alone until such time as a window update is received *)
$cb'.tt\_rexmt$
) **in**

(* Update the send queue and window *)
let $(snd\_wnd', sndq') =$ (**if** *ourfinisacked* **then**
(* If this socket has previously emitted a *FIN* segment and the *FIN* has now been *ACK*ed, decrease *snd_wnd* by the length of the send queue and clear the send queue.*)
$(cb'.snd\_wnd - \textbf{length } tcp\_sock\_0.sndq, [\,])$
else
(* Otherwise, reduce the send window by the amound of data acknowledged as it is now consuming space on the receiver's receive queue. Remove the acknowledged bytes from the send queue as they will never need to be retransmitted.*)
$(cb'.snd\_wnd - \textbf{num}(ack - tcp\_sock\_0.cb.snd\_una),$
$\text{DROP}(\textbf{num}(ack - tcp\_sock\_0.cb.snd\_una))tcp\_sock\_0.sndq)$
) **in**

(* Update the control block *)
modify_cb($\lambda cb.cb$
$\llparenthesis$ (* If *revert_rexmt* (above) flags that a bad retransmission occured, undo the congestion avoidance changes *)

snd_cwnd :$\hat{=}$ $cb.snd\_cwnd\_prev$ **onlywhen** $revert\_rexmt$;
$snd\_ssthresh$:$\hat{=}$ $cb.snd\_ssthresh\_prev$ **onlywhen** $revert\_rexmt$;
snd_nxt :$\hat{=}$ $cb'.snd\_max$ **onlywhen** $revert\_rexmt$;
$t\_badrxtwin$:$\hat{=}$ TimeWindowClosed **onlywhen** $revert\_rexmt$
$\rrbracket$) andThen
modify_cb($\lambda cb.cb$
$\llbracket$
(* Update the round-trip time estimates and retransmit timer *)
$t\_rttinf := t\_rttinf'$;
$tt\_rexmt := tt\_rexmt'$;

(* If the ACK segment allowed us to successfully time a segment (and update the round-trip time estimates) then clear the soft error flag and clear the segment round-trip timer in order that it can be used on a future segment. *)
$t\_softerror$:$\hat{=}$ $*$ **onlywhen is_some** $emission\_time$;
t_rttseg :$\hat{=}$ $*$ **onlywhen is_some** $emission\_time$;

(* Update the congestion window by the algorithm in expand_cwnd (p99) only when not performing NewReno retransmission or the duplicate ACK counter is zero, i.e., expand the congestion window when this ACK is not a NewReno-style partial ACK and hence the connection has yet recovered *)
snd_cwnd :$\hat{=}$ expand_cwnd $cb.snd\_ssthresh$ $tcp\_sock\_0.cb.t\_maxseg$
(TCP_MAXWIN $\ll$ $tcp\_sock\_0.cb.snd\_scale$) $cb.snd\_cwnd$
onlywhen($\neg$ TCP_DO_NEWRENO $\vee cb'.t\_dupacks = 0$);
$snd\_wnd := snd\_wnd'$; (* The updated send window *)
$snd\_una := ack$; (* Have had up to ack acknowledged *)
$snd\_nxt :=$ **max** ack $cb.snd\_nxt$; (* Ensure invariant $snd\_nxt \geq snd\_una$ *)

(* Reset the $2MSL$ timer if in the TIME_WAIT state as have received a valid ACK segment for the waiting socket *)
tt_2msl :$\hat{=}$ $\uparrow((())_{\text{slow\_timer}(2*\text{TCPTV\_MSL})})$
onlywhen($tcp\_sock\_0.st$ = TIME_WAIT)
$\rrbracket$) andThen
$modify\_tcp\_sock$($\lambda s.s$ $\llbracket$ $sndq := sndq'$ $\rrbracket$) andThen (* The send queue update *)

(**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 *)
modify_sock(tcp_close $arch$) andThen
stop
else if $tcp\_sock\_0.st$ = TIME_WAIT $\wedge$ $ack > tcp\_sock\_0.cb.snd\_una$(* data acked past FIN *) **then**
(* If the socket is in TIME_WAIT and this segment contains a new acknowledgement (that acknowledges past the FIN segment, drop it—it's invalid. Stop processing. *)
mlift_dropafterack_or_fail seg $arch$ $rttab$ $ifds$ $ticks$ andThen
stop
else
(* Otherwise, flag that $deliver\_in\_3$ can continue processing the segment if need be *)
cont)

)(* cb' *)

– $deliver\_in\_3$ **ACK processing :**
di3_ackstuff tcp_sock_0 seg $ourfinisacked$ $arch$ $rttab$ $ifds$ $ticks$ =
(* Pull some fields out of the segment *)
let $ack = tcp\_seq\_flip\_sense$ $seg.ack$ **in**
let $seq = tcp\_seq\_flip\_sense$ $seg.seq$ **in**
let $data = seg.data$ **in**

(* Pull out senders advertised window from the segment, applying the sender's scaling *)

let $win = \mathbf{w2n}\ seg.win \ll tcp\_sock\_0.cb.snd\_scale$ **in**

(* Get the socket's control block using the monadic state accessor get_cb. Process the acknowledgement data in the segment, do some congestion control calculations and finally update the control blocks *)
(get_cb λcb.
(* The segment is possibly a duplicate ack if it contains no data, does not contain a window update and the socket has unacknowledged data (the retransmit timer is still active). The no data condition is important: if this socket is sending little or no data at present and is waiting for some previous data to be acknowledged, but is receiving data filled segments from the other end, these may all contain the same acknowledgement number and trigger the retransmit logic erroneously. *)
let $has\_data = (data \neq [\,] \wedge$
$(\text{bsd\_arch}\ arch \implies (cb.rcv\_nxt < seq + \mathbf{length}\ data \wedge seq < cb.rcv\_nxt + cb.rcv\_wnd)))$ **in**
let $maybe\_dup\_ack = (\neg has\_data \wedge win = cb.snd\_wnd \wedge \text{mode\_of}\ cb.tt\_rexmt = \uparrow \text{REXMT})$ **in**

if $ack \leq cb.snd\_una \wedge maybe\_dup\_ack$ **then**
(* Received a duplicate acknowledgement: it is an old acknowledgement (strictly less than *snd_una*) and it meets the duplicate acknowledgement conditions above. Do Fast Retransmit/Fast Recovery Congestion Control (RFC 2581 Ch3.2 Pg6) and NewReno-style Fast Recovery (RFC 2582, Ch3 Pg3), updating the control block variables and creating segments for transmission as appropriate. *)

let $t\_dupacks' = cb.t\_dupacks + 1$ **in**

if $t\_dupacks' < 3$ **then**
(* Fewer than three duplicate acks received so far. Just increment the duplicate ack counter. We must continue processing, in case *FIN* is set. *)
$\text{modify\_cb}(\lambda cb'.cb' \llparenthesis\ t\_dupacks := t\_dupacks' \rrparenthesis)$ andThen
cont

else if $t\_dupacks' > 3 \vee (t\_dupacks' = 3 \wedge \text{TCP\_DO\_NEWRENO} \wedge ack < cb.snd\_recover)$ **then**
(* If this is the 4th or higher duplicate *ACK* then Fast Retransmit/Fast Recovery congestion control is already in progress. Increase the congestion window by another maximum segment size (as the duplicate *ACK* indicates another out-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". *)
$\text{modify\_cb}(\lambda cb'.cb' \llparenthesis\ t\_dupacks := \mathbf{if}\ t\_dupacks' = 3\ \mathbf{then}\ 0$ (* false retransmit, or further loss or delay *)
$\mathbf{else}\ t\_dupacks';$
$snd\_cwnd := cb.snd\_cwnd + cb.t\_maxseg \rrparenthesis)$ andThen
mlift_tcp_output_perhaps_or_fail *ticks arch rttab ifds* andThen
stop (* no need to process the segment any further *)

else if $t\_dupacks' = 3 \wedge \neg(\text{TCP\_DO\_NEWRENO} \wedge ack < cb.snd\_recover)$ **then**
(* If this is the 3rd duplicate segment and if the host supports NewReno extensions, a NewReno-style Fast Retransmit is not already in progress, then do a Fast Retransmit *)

(* Update the control block before the retransmit to reflect which data requires retransmission *)
$\text{modify\_cb}(\lambda cb'.cb' \llparenthesis\ t\_dupacks := t\_dupacks';$ (* increment the counter *)

(* Set to half the current flight size as per RFC2581/2582 *)
$snd\_ssthresh := \mathbf{max}\ 2((\mathbf{min}\ cb.snd\_wnd\ cb.snd\_cwnd)\ \mathbf{div}\ 2$
$\mathbf{div}\ cb.t\_maxseg) * cb.t\_maxseg;$

(* If doing NewReno-style Fast Retransmit set to the highest sequence number transmitted so far *snd_max*. *)
$snd\_recover :\hat{=}\ cb.snd\_max$ **onlywhen** TCP_DO_NEWRENO;

(* Clear the retransmit timer and round-trip time measurement timer. These will be started by tcp_output_really when the retransmit is actioned. *)
$tt\_rexmt := *$;
$t\_rttseg := *$;

(* Sequence number to retransmit—this is equal to the *ack* value in the duplicate *ACK* segment *)
$snd\_nxt := ack$;
(* Ensure the congestion window is large enough to allow one segment to be emitted *)
$snd\_cwnd := cb.t\_maxseg \rrparenthesis$) andThen

(* Attempt to create a segment for output using the modified control block (this is all a relational monad idiom) *)
mlift_tcp_output_perhaps_or_fail *ticks arch rttab ifds* andThen

(* Finally, update the congestion window to *snd_ssthresh* plus 3 maximum segment sizes (this is the artificial inflation of RFC2581/2582 because it is known that the 3 segments that generated the 3 duplicate acknowledgments are received and no longer consuming network resource. Also put *snd_nxt* back to its previous value. *)
modify_cb($\lambda cb'.cb' \llparenthesis\, snd\_cwnd := cb'.snd\_ssthresh + cb.t\_maxseg * t\_dupacks'$;
$snd\_nxt := \mathbf{max}\ cb.snd\_nxt\ cb'.snd\_nxt \rrparenthesis$) andThen
stop (* no need to process the segment any further *)

else *assert_failure*"di3_ackstuff" (* Believed to be impossible—here for completion and safety *)

else if $ack \leq cb.snd\_una \wedge \neg maybe\_dup\_ack$ **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. *)
modify_cb($\lambda cb'.cb' \llparenthesis\, t\_dupacks := 0 \rrparenthesis$)

else (* Must be: $ack > cb.snd\_una$ *)
(* This is the *ACK* of a new sequence number—this case is handled by the auxiliary function di3_newackstuff (p295) *)
di3_newackstuff *tcp_sock_0 seg ourfinisacked arch rttab ifds ticks*
)

– *deliver_in_3* **data processing :**
di3_datastuff_really the_ststuff tcp_sock_0 seg bsd_fast_path arch =
(* Pull some fields out of the segment *)
let $ACK = seg.ACK$ **in**
let $FIN = seg.FIN$ **in**
let $PSH = seg.PSH$ **in**
let $URG = seg.URG$ **in**
let $ack = tcp\_seq\_flip\_sense\ seg.ack$ **in**
let $urp = \mathbf{w2n}\ seg.urp$ **in**
let $data = seg.data$ **in**
let $seq = tcp\_seq\_flip\_sense\ seg.seq + (\mathbf{if}\ seg.SYN\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$ **in**

(* Pull out the senders advertised window and apply the sender's scale factor *)
let $win = \mathbf{w2n}\ seg.win \ll (tcp\_sock\_0).cb.snd\_scale$ **in**

(* Get the socket's control block using the monadic state accessor get_cb. Process the segments data and possibly update the send window *)
(get_sock $\lambda sock$.
let tcp_sock = tcp_sock_of *sock* **in**
let $cb = tcp\_sock.cb$ **in**

(* Trim segment to be within the receive window *)

(* Trim duplicate data from the left edge of *data*, i.e., data before *cb.rcv_nxt*. Adjust *seq*, *URG* and *urp* in respect of left edge trimming. If the urgent data has been trimmed from the segment's data, *URG* is cleared also. Note: the urgent pointer always points to the byte immediately following the urgent byte and is relative to the start of the segment's data. An urgent pointer of zero signifies that there is no urgent data in the segment. *)

$\mathbf{let}\ \ trim\_amt\_left = \mathbf{if}\ \ cb.rcv\_nxt > seq\ \ \mathbf{then}\ \ \mathbf{min}(\mathbf{num}(cb.rcv\_nxt - seq))(\mathbf{length}\ \ data)$
$\qquad\qquad\qquad\qquad \mathbf{else}\ \ 0\ \ \mathbf{in}$
$\mathbf{let}\ \ data\_trimmed\_left = \mathrm{DROP}\ \ trim\_amt\_left\ data\ \ \mathbf{in}$
$\mathbf{let}\ \ seq\_trimmed = seq + trim\_amt\_left\ \ \mathbf{in}$ (* Trimmed data starts at *seq_trimmed* *)
$\mathbf{let}\ \ urp\_trimmed = \mathbf{if}\ \ urp > trim\_amt\_left\ \ \mathbf{then}\ \ urp - trim\_amt\_left\ \ \mathbf{else}\ \ 0\ \ \mathbf{in}$
$\mathbf{let}\ \ URG\_trimmed = \mathbf{if}\ \ urp\_trimmed \neq 0\ \ \mathbf{then}\ \ URG\ \ \mathbf{else}\ \ \mathbf{F}\ \ \mathbf{in}$

(* Trim any data outside the receive window from the right hand edge. If all the data is within the window and the *FIN* flag is set then the *FIN* flag is valid and should be processed. Note: this trimming may remove urgent data from the segment. The urgent pointer and flag are not cleared here because there is still urgent data to be received, but now in a future segment. *)

$\mathbf{let}\ \ data\_trimmed\_left\_right = \mathrm{TAKE}\ \ cb.rcv\_wnd\ data\_trimmed\_left\ \ \mathbf{in}$
$\mathbf{let}\ \ FIN\_trimmed = \mathbf{if}\ \ data\_trimmed\_left\_right = data\_trimmed\_left\ \ \mathbf{then}\ \ FIN\ \ \mathbf{else}\ \ \mathbf{F}\ \ \mathbf{in}$

(* Processing of urgent (OOB) data: *)

(* We have a valid urgent pointer iff the trimmed segment has its urgent flag set with a non-zero urgent pointer, and the urgent pointer plus the length of the receive queue is less than or equal to SB_MAX. The last condition is imposed by FreeBSD, supposedly to prevent `soreceive` from crashing (although we cannot identify why it might crash). *)

$\mathbf{let}\ \ urp\_valid = (URG\_trimmed \wedge urp\_trimmed > 0 \wedge urp\_trimmed + \mathbf{length}\ \ tcp\_sock.rcvq \leq \mathrm{SB\_MAX})\ \mathbf{in}$

(* This is a new urgent pointer, i.e., it is greater than any previous one stored in *cb.rcv_up*. Note: the urgent pointer is relative to the sequence number of a segment *)

$\mathbf{let}\ \ urp\_advanced = (urp\_valid \wedge (seq\_trimmed + urp\_trimmed > cb.rcv\_up))\ \mathbf{in}$

(* The urgent pointer lies within segment *seq* and the socket is not set to do inline delivery, therefore it is possible to pull out the urgent byte from the stream *)

$\mathbf{let}\ \ can\_pull = (urp\_valid \wedge$
$\quad urp\_trimmed \leq \mathbf{length}\ \ data\_trimmed\_left\_right \wedge sock.sf.b(\mathrm{SO\_OOBINLINE}) = \mathbf{F})\ \mathbf{in}$

(* Build trimmed segment to place on reassembly queue. If urgent data is in this segment and the socket is not doing inline delivery (and hence the urgent byte is stored in *iobc*), remove the urgent byte from the segment's data so that it does not get placed in the receive queue, and set *spliced_urp* to the sequence number of the urgent byte. *)

$\mathbf{let}\ \ rseg = \langle\!|\ seq := seq\_trimmed;$
$\qquad spliced\_urp := \mathbf{if}\ \ can\_pull\ \ \mathbf{then}\ \ \uparrow(cb.rcv\_nxt + urp\_trimmed - 1)\ \mathbf{else}\ \ *;$
$\qquad FIN := FIN\_trimmed;$
$\qquad data := \mathbf{if}\ \ can\_pull\ \ \mathbf{then}$
$\qquad\quad (\mathrm{TAKE}(urp - 1)\,data\_trimmed\_left\_right)\ @\ (\mathrm{DROP}\ \ urp\ data\_trimmed\_left\_right)$
$\qquad\ \ \mathbf{else}\ \ data\_trimmed\_left\_right$
$\quad |\!\rangle\ \mathbf{in}$

(* Perform a monadic socket state update *)

$modify\_tcp\_sock(\lambda s.s$
$\qquad \langle\!|\ cb := s.cb$
$\qquad\ \ \langle\!|$ (* If the segment's urgent pointer is valid and advances the urgent pointer, update *rcv_up* with the new absolute pointer, otherwise just pull it along with the left hand edge of the receive window. Note: an earlier segment may have set *rcv_up* to point somewhere into a future segment. The use of **max** ensures that the pointer is not accidentally overwritten until the future segment arrives. *)
(* FreeBSD does not pull *rcv_up* along in the fast path; this is a bug *)
$\qquad rcv\_up\ :\hat{=}(\mathbf{if}\ \ urp\_advanced\ \ \mathbf{then}\ \ seq\_trimmed + urp\_trimmed$
$\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{max}\ \ cb.rcv\_up\ cb.rcv\_nxt)$
$\qquad\quad \mathbf{onlywhen}\,\neg(\mathrm{bsd\_arch}\ \ arch \wedge bsd\_fast\_path)|\!\rangle;$

(* If the urgent pointer is valid and advances the urgent pointer, update *rcvurp*—the socket's receive queue urgent data index—to be the index into the receive queue where the new urgent data will be stored. Note: the subtraction of 1 is correct because *rcvurp* points to the location where the urgent byte is stored not the byte immediately following the urgent byte (as is the convention for the *urp* field in the TCP header). *)

$$rcvurp :\hat{=} (\uparrow(\textbf{length}\ tcp\_sock.rcvq + \textbf{num}(seq\_trimmed + urp\_trimmed - cb.rcv\_nxt - 1))) \quad \textbf{onlywhen}\ urp\_advanced;$$

(* If the segment's urgent pointer is valid, the urgent data is within this segment and the socket is not doing inline delivery of urgent data, pull out the urgent byte into *iobc*. If the urgent data is within a future segment set *iobc* to NO_OOBDATA to signify that the urgent data is not available yet, otherwise leave *iobc* alone if the urgent pointer is not valid. *)

$$iobc :\hat{=} (\textbf{if}\ can\_pull\ \textbf{then}\ \mathrm{OOBDATA}(EL(urp - 1)\ data\_trimmed\_left\_right)\ \textbf{else}\ \mathrm{NO\_OOBDATA}) \quad \textbf{onlywhen}\ urp\_valid$$

$\rrbracket$) andThen

(* Processing of non-urgent data. There are 6 cases to consider: *)

(chooseM$\{\mathbf{F}; \mathbf{T}\}\lambda FIN_reass.$

(* Case (1) The segment contains new in-order, in-window data possibly with a *FIN* and the receive window is not closed. Note: it is possible that the segment contains just one byte of OOB data that may have already been pulled out into *iobc* if OOB delivery is out-of-line. In which case, the below must still be performed even though no data is contributed to the reassembly buffer in order that *rcv_nxt* is updated correctly (because a byte of urgent data consumes a byte of sequence number space). This is why *data_trimmed_left_right* is used rather than *data_deoobed* in some of the conditions below. *)

$$(\textbf{if}\ seq\_trimmed = cb.rcv\_nxt \wedge seq\_trimmed + \textbf{length}\ data\_trimmed\_left\_right + (\textbf{if}\ FIN\_trimmed\ \textbf{then}\ 1\ \textbf{else}\ 0) > cb.rcv\_nxt \wedge cb.rcv\_wnd > 0\ \textbf{then}$$

(* Only need to acknowledge the segment if there is new in-window data (including urgent data) or a valid *FIN* *)

$$\textbf{let}\ have\_stuff\_to\_ack = (data\_trimmed\_left\_right \neq [\,] \vee FIN\_trimmed)\ \textbf{in}$$

(* If the socket is connected, has data to *ACK* but no *FIN* to *ACK*, the reassembly queue is empty, the socket is not currently within a bad retransmit window and an *ACK* is not already being delayed, then delay the *ACK*. *)

$$\textbf{let}\ delay\_ack = (tcp\_sock.st \in \{\mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}; \mathrm{FIN\_WAIT\_1}; \mathrm{CLOSING}; \mathrm{LAST\_ACK}; \mathrm{FIN\_WAIT\_2}\} \wedge have\_stuff\_to\_ack \wedge \neg FIN\_trimmed \wedge cb.t\_segq = [\,] \wedge \neg cb.tf\_rxwin0sent \wedge cb.tt\_delack = *)\ \textbf{in}$$

(* Check to see whether any data or a *FIN* can be reassembled. tcp_reass returns the set of all possible reassemblies, one of which is chosen non-deterministically here. Note: a *FIN* can only be reassembled once all the data has been reassembled. The *len* result from tcp_reass is the length of the reassembled data, *data_reass*, plus the length of any out-of-line urgent data that is not included in the reassembled data but logically occurs within it. This is to ensure that control block variables such as *rcv_nxt* are incremented by the correct amount, i.e., by the amount of data (whether urgent or not) received successfully by the socket. See tcp_reass (p100) for further details. *)

$$\textbf{let}\ rsegq = rseg :: cb.t\_segq\ \textbf{in}$$

$$(\mathrm{chooseM}(\mathrm{tcp\_reass}\ cb.rcv\_nxt\ rsegq)\lambda(data\_reass, len, FIN\_reass0).$$

(* Length (in sequence space) of reassembled data, counting a *FIN* as one byte and including any out-of-line urgent data previously removed *)

$$\textbf{let}\ len\_reass = len + (\textbf{if}\ FIN\_reass0\ \textbf{then}\ 1\ \textbf{else}\ 0)\ \textbf{in}$$

(* Add the reassembled data to the receive queue and increment *rcv_nxt* to mark the sequence number of the byte past the last byte in the receive queue*)

$$\textbf{let}\ rcvq' = tcp\_sock.rcvq\ @\ data\_reass\ \textbf{in}$$

let $rcv\_nxt' = cb.rcv\_nxt + len\_reass$ **in** (* includes oob bytes as they occupy sequence space *)

(* Prune the receive queue of any data or *FIN*s that were reassembled, keeping all segments that contain data at or past sequence number $cb.rcv\_nxt + len\_reass$. *)
let $t\_segq' =$ tcp_reass_prune $rcv\_nxt'\ rsegq$ **in**

(* Reduce the receive window in light of the data added to the receive queue. Do not include out-of-line urgent data because it does not store data in the receive queue. *)
let $rcv\_wnd' = cb.rcv\_wnd -$ **length** $data\_reass$ **in**

(* Hack: assertion used to share values with later conditions *)
assert($FIN\_reass = FIN\_reass0$) andThen

(* Update the socket state *)
$modify\_tcp\_sock(\lambda s.s$
$\lparr rcvq := rcvq'$; (* the updated receive queue *)
$cb := s.cb$
$\lparr$ (* Start the delayed ack timer if decided to earlier, i.e., $delay\_ack = \mathbf{T}$. *)
$tt\_delack \mathrel{:\hat{=}} \uparrow((())_{\text{fast\_timer}}\ \text{TCPTV\_DELACK})$ **onlywhen** $delay\_ack$;
(* Set if not delaying an *ACK* and have stuff to *ACK* *)
$tf\_shouldacknow \mathrel{:\hat{=}} \neg delay\_ack$ **onlywhen** $have\_stuff\_to\_ack$;
$t\_segq := t\_segq'$; (* updated reassembly queue, post-pruning *)
$rcv\_nxt := rcv\_nxt'$;
$rcv\_wnd := rcv\_wnd'$
$\rparr$
$\rparr)$
)(* chooseM *)

(* Case (2) The segment contains new out-of-order in-window data, possibly with a *FIN*, and the receive window is not closed. Note: it may also contain in-window urgent data that may have been pulled out-of-line but still require processing to keep reassembly happy. *)
else if $seq\_trimmed > cb.rcv\_nxt \land seq\_trimmed < cb.rcv\_nxt + cb.rcv\_wnd \land$
length $data\_trimmed\_left\_right +$ (**if** $FIN\_trimmed$ **then** 1 **else** 0) $> 0 \land$
$cb.rcv\_wnd > 0$ **then**

(* Hack: assertion used to share values with later conditions *)
assert($FIN\_reass = \mathbf{F}$) andThen

(* Update the socket's TCP control block state *)
modify_cb($\lambda cb.cb\ \lparr$ (* Add the segment to the reassembly queue *)
$t\_segq := rseg :: cb.t\_segq$;
(* Acknowledge out-of-order data immediately (per RFC2581 Ch4.2) *)
$tf\_shouldacknow := \mathbf{T}$
$\rparr)$

(* Case (3) The segment is a pure *ACK* segment (contains no data) (and must be in-order). *)
(* Invariant here that $seq\_trimmed = seq$ if segment is a pure *ACK*. Note: the length of the original segment (not the trimmed segment) is used in the guard to ensure this really was a pure *ACK* segment. *)
else if $ACK \land seq\_trimmed = cb.rcv\_nxt \land$ **length** $data +$ (**if** FIN **then** 1 **else** 0) $= 0$ **then**

(* Hack: assertion used to share values with later conditions *)
assert($FIN\_reass = \mathbf{F}$) (* Have not received a FIN *)

(* Case (4) Segment contained no useful data—was a completely old segment. Note: the original fields from the segment, i.e., *seq*, *data* and *FIN* are used in the guard below—the trimmed variants are useless here! *)
(* Case (5) Segment is a window probe. Note: the original fields from the segment, i.e., *data* and *FIN* are used in the guard below—the trimmed variants are useless here! *)
(* Case (6) Segment is completely beyond the window and is not a window probe *)
else if ($seq < cb.rcv\_nxt \land seq +$ **length** $data +$ (**if** FIN **then** 1 **else** 0) $\leq cb.rcv\_nxt) \lor$ (* (4) *)
$(seq\_trimmed = cb.rcv\_nxt \land cb.rcv\_wnd = 0 \land$

length $data + ($**if** FIN **then** 1 **else** $0) > 0) \lor$ (* (5) *)
T **then** (* (6) *)

(* Hack: assertion used to share values with later conditions *)
assert($FIN\_reass = \mathbf{F}$) andThen (* Definitely false—segment is outside window *)

(* Update socket's control block to assert that an ACK segment should be sent now. *)
(* Source: TCPIPv2p959 says "segment is discarded and an ack is sent as a reply" *)
modify_cb($\lambda cb.cb$ ⦇ $tf\_shouldacknow := \mathbf{T}$⦈)

else
assert_failure"di3_datastuff"(* impossible *)

) andThen

(* Finished processing the segment's data *)
(* Thread the reassembled FIN flag through to di3_ststuff *)
the_ststuff FIN_reass

)(* chooseM FIN_reass *)
)(* get_sock \sock *)

– *deliver_in_3* **data processing :**
di3_datastuff *the_ststuff tcp_sock_0 seg ourfinisacked arch* =
(* Pull some fields out of the segment *)
let $ACK = seg.ACK$ **in**
let $FIN = seg.FIN$ **in**
let $PSH = seg.PSH$ **in**
let $URG = seg.URG$ **in**
let $ack = tcp\_seq\_flip\_sense\ seg.ack$ **in**
let $urp = $ **w2n** $seg.urp$ **in**
let $data = seg.data$ **in**
let $seq = tcp\_seq\_flip\_sense\ seg.seq + ($**if** $seg.SYN$ **then** 1 **else** $0)$ **in**
let $win = $ **w2n** $seg.win \ll (tcp\_sock\_0).cb.snd\_scale$ **in**

get_sock $\lambda sock$.
let $tcp_sock = $ tcp_sock_of $sock$ **in**
let $cb = tcp\_sock.cb$ **in**

(* Various things do not happen if BSD processes the segment using its header prediction (fast-path) code. Header prediction occurs only in the ESTABLISHED state, with segments that have only ACK and/or PSH flags set, are in-order, do not contain a window update, when data is not being retransmitted (no congestion is occuring) and either: (a) the segment is a valid pure ACK segment of new data, less than three duplcicate ACKs have been received and the congestion window is at least as large as the send window, or (b) the segment contains new data, does not acknowlegdge any new data, the segment reassembly queue is empty and there is space for the segment's data in the socket's receive buffer. *)
let $bsd\_fast\_path = ((tcp\_sock.st = \mathrm{ESTABLISHED}) \land \neg seg.SYN \land \neg FIN \land \neg seg.RST \land$
$\neg URG \land ACK \land seq = cb.rcv\_nxt \land cb.snd\_wnd = win \land$
$cb.snd\_max = cb.snd\_nxt \land ($
$(ack > cb.snd\_una \land ack \leq cb.snd\_max \land$
$cb.snd\_cwnd \geq cb.snd\_wnd \land cb.t\_dupacks < 3)$
$\lor$
$(ack = cb.snd\_una \land cb.t\_segq = [\,] \land$
$($**length** $data) <$
$(sock.sf.n(\mathrm{SO_RCVBUF}) - $ **length** $tcp\_sock.rcvq))))$ **in**

(* Update the send window using the received segment if the segment will not be processed by BSD's fast path, has the ACK flag set, is not to the right of the window, and either:
(a) the last window update was from a segment with sequence number less than *seq*, i.e., an older segment than the current segment, or
(b) the last window update was from a segment with sequence number equal to *seq* but with an acknowledgement number less than *ack*, i.e., this segment acknowledges newer data than the segment the last window update was taken from, or
(c) the last window update was from a segment with sequence number equal to *seq* and acknowledgement number equal to *ack*, i.e., a segment similar to that the previous update came from, but this segment contains a larger window advertisment than was previously advertised, or
(d) this segment is the third segment during connection establishement (state is SYN_RECEIVED) and does not have the *FIN* flag set. *)

let $update\_send\_window = (\neg bsd\_fast\_path \wedge seg.ACK \wedge seq \leq cb.rcv\_nxt + cb.rcv\_wnd \wedge$
$(cb.snd\_wl1 < seq \vee$
$(cb.snd\_wl1 = seq \wedge$
$(cb.snd\_wl2 < ack \vee cb.snd\_wl2 = ack \wedge win > cb.snd\_wnd)) \vee$
$(tcp\_sock.st = \text{SYN\_RECEIVED} \wedge \neg FIN)))$ **in** (* This replaces BSD's `snd_wl1 := seq-1` hack; should perhaps be $\neg FIN\_reass$ *)

let $seq\_trimmed = \mathbf{max}\ seq(\mathbf{min}\ cb.rcv\_nxt(seq + \mathbf{length}\ data))$ **in**

(* Write back the window updates *)
modify_cb($\lambda cb.cb$ ⦇ $snd\_wnd \mathrel{:\hat{=}} win$ **onlywhen** $update\_send\_window$;
$snd\_wl1 \mathrel{:\hat{=}} seq\_trimmed$ **onlywhen** $update\_send\_window$;
$snd\_wl2 \mathrel{:\hat{=}} ack$ **onlywhen** $update\_send\_window$
(* persist timer will be set by *deliver_out_1* if this updates the window to zero and there is data to send *)
⦈) andThen

(* If in TIME_WAIT or will transition to it from CLOSING, ignore any URG, data, or FIN. Note that in FIN_WAIT_1 or FIN_WAIT_2, we still process data, even if *ourfinisacked*. *)
if $tcp\_sock.st = \text{TIME\_WAIT} \vee (tcp\_sock.st = \text{CLOSING} \wedge ourfinisacked)$ **then**
(* pull along urgent pointer *)
modify_cb($\lambda cb.cb$ ⦇ $rcv\_up := \mathbf{max}\ cb.rcv\_up\ cb.rcv\_nxt$⦈) andThen
the_ststuff **F**
else
di3_datastuff_really the_ststuff tcp_sock_0 seg bsd_fast_path arch

---

– *deliver_in_3* **TCP state change processing :**
di3_ststuff *FIN_reass ourfinisacked ack* =

(* The entirety of this function is an encoding of the TCP State Transition Diagram (as it is, not as it is traditionally depicted) post-SYN_SENT state. It specifies for given start state and set of conditions (all or some of which are affected by the processing of the current segment), which state the TCP socket should be moved into next *)

(* Get the TCP socket using the monadic state accessor get_cb. *)
(get_sock $\lambda sock$.
let cb = (tcp_sock_of $sock$).cb **in** (* ...and its control block *)

(* Several of the encoded transitions (below) require the socket to be moved into the TIME_WAIT state, in which case the `2MSL` timer is started, all other timers are cancelled and the socket's state is changed to TIME_WAIT. This common idiom is defined monadically as a function here *)
let *enter_TIME_WAIT* =
modify_tcp_sock($\lambda s.s$
⦇ $st :=$ TIME_WAIT;
$cb := s.cb$

$\langle\!|$ *tt_2msl* $:= \uparrow((())_{\text{slow\_timer}(2*\text{TCPTV\_MSL})})$;
tt_rexmt $:= *$;
tt_keep $:= *$;
tt_delack $:= *$;
tt_conn_est $:= *$;
tt_fin_wait_2 $:= *$
$|\!\rangle$
$|\!\rangle$) **in**

(* If the processing of the current segment has led to *FIN_reass* being asserted then the whole data stream from the other end has been received and reconstructed, including the final *FIN* flag. The socket should have its read-half flagged as shut down, i.e., *cantrcvmore* = **T**, otherwise the socket is not modified. *)
(**if** *FIN_reass* **then**
modify_sock($\lambda s.s$ $\langle\!|$ *cantrcvmore* := **T**$|\!\rangle$)
else cont) andThen

(* State Transition Diagram encoding: *)
(* The state transition encoding, case-split on the current state and whether a *FIN* from the remote end has been reassembled *)
case ((tcp_sock_of *sock*).*st*, *FIN_reass*) **of**

(SYN_RECEIVED, **F**) $\rightarrow$ (* In SYN_RECEIVED and have not received a *FIN* *)
if $ack \geq cb.iss + 1$ **then**
(* This socket's initial *SYN* has been acknowledged *)
modify_tcp_sock($\lambda s.s$
$\langle\!|$ *st* := **if** $\neg$*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
$|\!\rangle$)
else
(* Not a valid path *)
stop ‖

(SYN_RECEIVED, **T**) $\rightarrow$ (* In SYN_RECEIVED and have received a *FIN* *)
(* Enter the CLOSE_WAIT state, missing out ESTABLISHED *)
modify_tcp_sock($\lambda s.s$ $\langle\!|$ *st* := CLOSE_WAIT$|\!\rangle$) ‖

(ESTABLISHED, **F**) $\rightarrow$ (* In ESTABLISHED and have not received a *FIN* *)
(* Doing common-case data delivery and acknowledgements. Remain in ESTABLISHED. *)
cont ‖

(ESTABLISHED, **T**) $\rightarrow$ (* In ESTABLISHED and received a *FIN* *)
(* Move into the CLOSE_WAIT state *)
modify_tcp_sock($\lambda s.s$ $\langle\!|$ *st* := CLOSE_WAIT$|\!\rangle$) ‖

(CLOSE_WAIT, **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 ‖

$(\text{CLOSE\_WAIT}, \mathbf{T}) \rightarrow$ (* 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 ‖

$(\text{FIN\_WAIT\_1}, \mathbf{F}) \rightarrow$ (* 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. *)
$modify\_tcp\_sock(\lambda s.s$
$\langle\!\langle\ st := \text{FIN\_WAIT\_2};$
$cb := s.cb$
$\langle\!\langle\ tt\_fin\_wait\_2 \ \hat{:=}\ \uparrow((())_{\text{slow\_timer TCPTV\_MAXIDLE}})$
onlywhen $sock.cantrcvmore$ (* believe always true *)
$\rangle\!\rangle$
$\rangle\!\rangle)$

else
(* If this socket's *FIN* has not been acknowledged then remain in FIN_WAIT_1 *)
cont ‖

$(\text{FIN\_WAIT\_1}, \mathbf{T}) \rightarrow$ (* 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 *)
$modify\_tcp\_sock(\lambda s.s\ \langle\!\langle\ st := \text{CLOSING}\rangle\!\rangle)$ ‖

$(\text{FIN\_WAIT\_2}, \mathbf{F}) \rightarrow$ (* 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 ‖

$(\text{FIN\_WAIT\_2}, \mathbf{T}) \rightarrow$ (* In FIN_WAIT_2 and have received a *FIN* *)
(* Connection has been shutdown so enter TIME_WAIT *)
enter_TIME_WAIT ‖

$(\text{CLOSING}, \mathbf{F}) \rightarrow$ (* 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, **T**) → (* In CLOSING and have received a *FIN* *)
(* The received *FIN* is a duplicate *FIN* with a new sequence number so as per RFC793 is ignored – if it were a duplicate with the same sequence number as the previously accepted *FIN*, then the *deliver_in_3* acknowledgement processing function di3_ackstuff would have dropped it. *)
if *ourfinisacked* **then**
(* If this socket's *FIN* has been acknowledged then the connection is now successfully closed, so enter TIME_WAIT state *)
enter_TIME_WAIT
else
(* Otherwise, ignore the new *FIN* and remain in the same state *)
cont ‖

(LAST_ACK, **F**) → (* In LAST_ACK and have not received a *FIN* *)
(* Remain in LAST_ACK until this socket's *FIN* is acknowledged. Note: eventually the retransmit timer will fire forcing the *FIN* to be retransmitted. *)
cont ‖

(LAST_ACK, **T**) → (* In LAST_ACK and have received a *FIN* *)
(* This transition is handled specially at the end of di3_newackstuff at which point processing stops, thus this transition is not possible *)
assert_failure"di3_ststuff" (* impossible *) ‖

(TIME_WAIT, **F**) → (* In TIME_WAIT and have not received a *FIN* *)
(* Remaining in TIME_WAIT until the `2MSL` timer expires *)
cont ‖

(TIME_WAIT, **T**) → (* In TIME_WAIT and have received a *FIN* *)
(* Remaining in TIME_WAIT until the `2MSL` timer expires *)
cont
)

– *deliver_in_3* **socket update processing :**
di3_socks_update sid *socks* *socks'* =

let *sock_1* = *socks*[sid] **in**
$\exists$*tcp_sock_1*.
TCP_PROTO(*tcp_sock_1*) = *sock_1.pr* $\wedge$

(* 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* (p292) for further details. *)

let *interesting* = λ*sid'*.
sid' $\neq$ sid $\wedge$
case (*socks*[*sid'*]).*pr* **of**
UDP_PROTO *udp_sock* → **F**
‖ TCP_PROTO(*tcp_sock'*) →
case *tcp_sock'*.*lis* **of**
$*$ → **F**
‖ ↑ *lis* →
sid $\in$ *lis*.q_0 **in**

let *interesting_sids* = (**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 $lis.q_0$. *)
$\exists sid'\ sock'\ tcp\_sock'\ lis\ q0L\ q0R.$
$sid' \in interesting\_sids \wedge$
$sock' = socks[sid'] \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}\ tcp\_sock' \wedge$
$sid' \neq \mathsf{sid} \wedge$
$tcp\_sock'.lis = \uparrow lis \wedge$
$lis.q_0 = q0L\ @\ (\mathsf{sid} :: q0R) \wedge$

(* Choose non-deterministically whether there is room on the queue of completed connections *)
choose $ok :: \mathrm{accept\_incoming\_q}\ lis.$

if ok **then**
(* If there is room, then remove socket sid from the queue of incomplete connections and add it to the queue of completed connections. *)
let $lis' = lis\ (\!|\ q_0 := q0L\ @\ q0R;$
$q := \mathsf{sid} :: lis.q |\!)$ **in**

(* Update the newly connected sockets receive window *)
let $rcv\_window = \mathrm{calculate\_bsd\_rcv\_wnd}\ sock\_1.sf\ tcp\_sock\_1$ **in**
(* BSD bug - rcv_adv gets incorrectly set using the old value of rcv_wnd, as this is done by the syncache, which is called from `tcp_input()` before the rcv_wnd update takes place. Note that we have the following: `SYN_SENT->ESTABLISHED =>` update rcv_wnd then rcv_adv `SYN_RCVD->ESTABLISHED =>` update rcv_adv then rcv_wnd *)
let $cb' = tcp\_sock\_1.cb\ (\!|\ rcv\_wnd := rcv\_window;$
$rcv\_adv := tcp\_sock\_1.cb.rcv\_nxt + tcp\_sock\_1.cb.rcv\_wnd |\!)$ **in**

(* Update both the newly connected socket and the listening socket *)
$socks' = socks \oplus$
$[(\mathsf{sid}, sock\_1\ (\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock\_1\ (\!|\ cb := cb' |\!)) |\!));$
$(sid', sock'\ (\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock'\ (\!|\ lis := \uparrow lis' |\!)) |\!))]$

else
(* ...otherwise there is no room on the listening socket's completed connections queue, so drop the newly connected socket and remove it from the listening socket's queue of incomplete connections. Note: the dropped connection is not sent a RST but a RST is sent upon receipt of further segments from the other end as the socket entry has gone away. *)
(* Note that the above note needs to be verified by testing. *)
let $lis' = lis\ (\!|\ q_0 := q0L\ @\ q0R |\!)$ **in**
$socks' = socks \oplus (sid', sock'\ (\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock'\ (\!|\ lis := \uparrow lis' |\!)) |\!))$

else
(* There is no such socket with socket sid on its queue of incomplete connections, thus socket sid was involved in a simultaneous open. Do not update any socket. *)
$socks' = socks$

---

deliver_in_3a **tcp: network nonurgent** **Receive data with invalid checksum or offset**

$h\ \langle\!|socks := socks;\ iq := iq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks;\ iq := iq'|\!\rangle$

(* **Summary:** This rule is a placeholder for the case where a received segment has an invalid checksum or offset, in which case implementations should drop it on the floor. The model of TCP segments does not contain checksum or offset, however, hence the **F** below. *)

$sid \in \mathbf{dom}(socks) \wedge$
$sock\_0 = socks[sid] \wedge$
$sock\_0.is_1 = \uparrow i_1 \wedge sock\_0.ps_1 = \uparrow p_1 \wedge sock\_0.is_2 = \uparrow i_2 \wedge sock\_0.ps_2 = \uparrow p_2 \wedge$

$sock\_0.pr = \mathrm{TCP\_PROTO}(tcp\_sock\_0) \wedge$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

$(\exists win\_\ urp\_\ ws\_discard\ mss\_discard.$
$win = \mathbf{w2n}\ win\_ \ll tcp\_sock\_0.cb.snd\_scale \wedge$
$urp = \mathbf{w2n}\ urp\_ \wedge$
$seg = \langle\!|$
$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq : \mathrm{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack : \mathrm{tcp\_seq\_local});$
$URG := URG;$
$ACK := ACK;$
$PSH := PSH;$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win\_;$
$ws := ws\_discard;$
$urp := urp\_;$
$mss := mss\_discard;$
$ts := ts;$
$data := data$
$|\!\rangle) \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly *)

$tcp\_sock\_0.st \notin \{\mathrm{CLOSED; LISTEN; SYN\_SENT}\} \wedge$
$tcp\_sock\_0.st \in \{\mathrm{SYN\_RECEIVED; ESTABLISHED; CLOSE\_WAIT; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSING; LAST\_ACK; TIME\_WAIT}\} \wedge$

F (* invalid checksum or offset *)

deliver_in_3b **tcp: network nonurgent** **Receive data after process has gone away**

$h\ \langle\!|socks := socks;\ iq := iq;\ oq := oq;\ bndlm := bndlm|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks';\ iq := iq';\ oq := oq';\ bndlm := bndlm'|\!\rangle$

(* **Summary:** if data arrives after the process associated with a socket has gone away, close socket and emit RST segment. *)

$sid \in \mathbf{dom}(socks) \wedge$
$sock\_0 = socks[sid] \wedge$
$sock\_0.is_1 = \uparrow i_1 \wedge sock\_0.ps_1 = \uparrow p_1 \wedge sock\_0.is_2 = \uparrow i_2 \wedge sock\_0.ps_2 = \uparrow p_2 \wedge$
$sock\_0.pr = \mathrm{TCP\_PROTO}(tcp\_sock\_0) \wedge$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

$(\exists win\_\ urp\_\ ws\_discard\ mss\_discard.$
$win = \mathbf{w2n}\ win\_ \ll tcp\_sock\_0.cb.snd\_scale \wedge$
$urp = \mathbf{w2n}\ urp\_ \wedge$

$seg = \langle\!|$
$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq : \mathrm{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack : \mathrm{tcp\_seq\_local});$
$URG := URG;$
$ACK := ACK;$
$PSH := PSH;$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win\_;$
$ws := ws\_discard;$
$urp := urp\_;$
$mss := mss\_discard;$
$ts := ts;$
$data := data$
$|\!\rangle) \wedge$

(* Note that there does not exist a better socket match to which the segment should be sent, as the whole quad is matched exactly. *)

(* test that this is data arriving after process has gone away *)
$tcp\_sock\_0.st \in \{\mathrm{FIN\_WAIT\_1; CLOSING; LAST\_ACK; FIN\_WAIT\_2; TIME\_WAIT}\} \wedge$
$sock\_0.fid = * \wedge$
$seq + \mathbf{length}\ data > tcp\_sock\_0.cb.rcv\_nxt \wedge$

(* close socket and emit RST segment *)
$socks' = socks \oplus (sid, \mathrm{tcp\_close}\ h.arch\ sock\_0) \wedge$
$\mathrm{dropwithreset\_ignore\_fail}\ seg\ h.arch\ h.ifds\ h.rttab(\mathrm{ticks\_of}\ h.ticks)$
$\mathrm{BANDLIM\_UNLIMITED}\ bndlm\ bndlm'\ outsegs \wedge$
$\mathrm{enqueue\_oq\_list\_qinfo}(oq, outsegs, oq')$

---

deliver_in_3c **tcp: network nonurgent** **Receive stupid ACK or LAND DoS in SYN_RECEIVED state**

$h\ \langle\!| socks := socks; \quad \xrightarrow{\tau} \quad h\ \langle\!| socks := socks';$
$iq := iq; \quad iq := iq';$
$oq := oq; \quad oq := oq';$
$bndlm := bndlm |\!\rangle \quad bndlm := bndlm' |\!\rangle$

(* **Summary:** if we receive a stupid ACK or a LAND DoS in SYN_RECEIVED state then update timers and emit a RST appropriately. *)

$sid \in \mathbf{dom}(socks) \wedge$
$sock\_0 = socks[sid] \wedge$
$sock\_0.is_1 = \uparrow i_1 \wedge sock\_0.ps_1 = \uparrow p_1 \wedge sock\_0.is_2 = \uparrow i_2 \wedge sock\_0.ps_2 = \uparrow p_2 \wedge$
$sock\_0.pr = \mathrm{TCP\_PROTO}(tcp\_sock\_0) \wedge$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

$(\exists win\_\ urp\_\ ws\_discard\ mss\_discard.$
$win = \mathbf{w2n}\ win\_ \ll tcp\_sock\_0.cb.snd\_scale \wedge$
$urp = \mathbf{w2n}\ urp\_ \wedge$
$seg = \langle\!|$

$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := \mathit{tcp\_seq\_flip\_sense}(seq : \text{tcp\_seq\_foreign});$
$ack := \mathit{tcp\_seq\_flip\_sense}(ack : \text{tcp\_seq\_local});$
$URG := URG;$
$ACK := ACK;$
$PSH := PSH;$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := FIN;$
$win := win\_;$
$ws := ws\_discard;$
$urp := urp\_;$
$mss := mss\_discard;$
$ts := ts;$
$data := data$
$\rrparenthesis) \wedge$

(* 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 for stupid ACK in SYN_RECEIVED, and for LAND DoS attack *)
$tcp\_sock\_0.st = \text{SYN\_RECEIVED} \wedge$
$((ACK \wedge (ack \leq tcp\_sock\_0.cb.snd\_una \vee ack > tcp\_sock\_0.cb.snd\_max)) \vee$
$seq < tcp\_sock\_0.cb.irs) \wedge$

(* incoming segment; update timers *)
let $(t\_idletime', tt\_keep', tt\_fin\_wait\_2') = \text{update\_idle } tcp\_sock\_0$ **in**
let $tcp\_sock' = tcp\_sock\_0 \llparenthesis cb := tcp\_sock\_0.cb \llparenthesis t\_idletime := t\_idletime'; tt\_keep := tt\_keep'; tt\_fin\_wait\_2 := tt\_fin\_wait\_2' \rrparenthesis\rrparenthesis$ **in**
$socks' = socks \oplus (sid, sock\_0 \llparenthesis pr := \text{TCP\_PROTO}(tcp\_sock')\rrparenthesis) \wedge$

(* emit RST. See dropwithreset_ignore_fail (p120) and enqueue_oq_list_qinfo (p**??**). *)
dropwithreset_ignore_fail $seg\ h.arch\ h.ifds\ h.rttab(\text{ticks\_of } h.ticks)$
BANDLIM_UNLIMITED $bndlm\ bndlm'\ outsegs \wedge$
enqueue_oq_list_qinfo$(oq, outsegs, oq')$

---

deliver_in_4 **tcp: network nonurgent** **Receive and drop (silently) a non-sane or martian segment**

$h \llparenthesis iq := iq \rrparenthesis \xrightarrow{\tau} h \llparenthesis iq := iq' \rrparenthesis$

(* **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$(iq, iq', \uparrow(\text{TCP } seg)) \wedge$
$seg.is_2 = \uparrow i_2 \wedge$
$is_1 = seg.is_1 \wedge$
$i_2 \in \text{local\_ips}(h.ifds) \wedge$
$(\mathbf{F} \vee$ (* 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**

$h\ \langle\!|iq := iq;\ oq := oq;\ bndlm := bndlm|\!\rangle \xrightarrow{\tau} h\ \langle\!|iq := iq';\ oq := oq';\ bndlm := bndlm'|\!\rangle$

(* **Summary:** Receive and drop any segment for this host that does not match any sockets (but does have sensible checksum and offset fields). Typically, generate RST in response, computing *ack* and *seq* to supposedly make the other end see this as an 'acceptable ack'. *)

dequeue_iq$(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

$seg.is_2 = \uparrow i_1 \wedge i_1 \in$ local_ips$(h.ifds) \wedge$
$seg.ps_2 = \uparrow p_1 \wedge$
$seg.is_1 \neq * \wedge seg.ps_1 \neq * \wedge$

T$\wedge$ (* placeholder for segment checksum and offset field sensible *)

$\neg(\exists((\mathsf{sid}, sock) :: h.socks) tcp\_sock.$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
match_score$(sock.is_1, sock.ps_1, sock.is_2, sock.ps_2)$
$(\mathbf{the}\ seg.is_1, seg.ps_1, \mathbf{the}\ seg.is_2, seg.ps_2) > 0$
$) \wedge$

dropwithreset $seg\ h.ifds$(ticks_of $h.ticks$)BANDLIM_RST_CLOSEDPORT $bndlm\ bndlm'\ outsegs' \wedge$
enqueue_and_ignore_fail $h.arch\ h.rttab\ h.ifds\ outsegs'\ oq\ oq'$

deliver_in_6 **tcp: network nonurgent** **Receive and drop (silently) a sane segment that matches a CLOSED socket**

$h\ \langle\!|iq := iq|\!\rangle \xrightarrow{\tau} h\ \langle\!|iq := iq'|\!\rangle$

(* **Summary:** Receive and drop any segment for this host that does not match any sockets (but does have sensible checksum or offset fields).
Note that pathological segments where is_1, ps_1, or ps_2 are not set in the segment are not dealt with here but need to be. *)

dequeue_iq$(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$
$(\exists((\mathsf{sid}, sock) :: h.socks) tcp\_sock.$
$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
match_score$(sock.is_1, sock.ps_1, sock.is_2, sock.ps_2)$
$(\mathbf{the}\ seg.is_1, seg.ps_1, \mathbf{the}\ seg.is_2, seg.ps_2) > 0 \wedge$
tcp_socket_best_match $h.socks(\mathsf{sid}, sock) seg\ h.arch \wedge$
$tcp\_sock.st = \mathrm{CLOSED}) \wedge$
$seg.is_2 = \uparrow i_1 \wedge i_1 \in$ local_ips$(h.ifds) \wedge$
T (* placeholder for segment checksum and offset field sensible *)

deliver_in_7 **tcp: network nonurgent Receive RST and zap non-{CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} socket**

$$h\ \langle\!|ts := ts \oplus (tid \mapsto (ts_{st})_d); \quad \xrightarrow{\tau} \quad h\ \langle\!|ts := ts \oplus (tid \mapsto (ts_{st})_d);$$
$$socks := socks \oplus [(sid, sock)]; \qquad socks := socks \oplus [(sid, sock')];$$
$$iq := iq|\!\rangle \qquad iq := iq'|\!\rangle$$

(* **Summary:** receive RST and silently zap non-{CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT} socket *)

dequeue_iq$(iq, iq', \uparrow(\text{TCP } seg)) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$
$st \notin \{\text{CLOSED; LISTEN; SYN\_SENT; SYN\_RECEIVED; TIME\_WAIT}\} \wedge$

$(\exists seq\_discard\ ack\_discard\ URG\_discard\ ACK\_discard\ PSH\_discard\ SYN\_discard\ FIN\_discard$
$\quad win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard.$
$seg = \langle\!|$
$\quad is_1 := \uparrow i_2;$
$\quad is_2 := \uparrow i_1;$
$\quad ps_1 := \uparrow p_2;$
$\quad ps_2 := \uparrow p_1;$
$\quad seq := tcp\_seq\_flip\_sense(seq\_discard : \text{tcp\_seq\_foreign});$
$\quad ack := tcp\_seq\_flip\_sense(ack\_discard : \text{tcp\_seq\_local});$
$\quad URG := URG\_discard;$
$\quad ACK := ACK\_discard;$
$\quad PSH := PSH\_discard;$
$\quad RST := \mathbf{T};$
$\quad SYN := SYN\_discard;$
$\quad FIN := FIN\_discard;$
$\quad win := win\_discard;$
$\quad ws := ws\_discard;$
$\quad urp := urp\_discard;$
$\quad mss := mss\_discard;$
$\quad ts := ts\_discard;$
$\quad data := data\_discard$
$\quad |\!\rangle$
$) \wedge$

((* $sock.st \in \{\text{CLOSED; LISTEN; SYN\_SENT; SYN\_RECEIVED; TIME\_WAIT}\}$ excluded already above *)
if $st \in \{\text{ESTABLISHED; FIN\_WAIT\_1; FIN\_WAIT\_2; CLOSE\_WAIT}\}$ **then**
$\quad err = \uparrow \text{ECONNRESET}$
else (* $sock.st \in \{\text{CLOSING; LAST\_ACK}\}$ – leave existing error *)
$\quad err = sock.es) \wedge$

(* see tcp_close (p121) *)
$sock' = \text{tcp\_close}\ h.arch(sock\ \langle\!|\ es := err|\!\rangle)$

deliver_in_7a **tcp: network nonurgent** **Receive RST and zap SYN_RECEIVED socket**

$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ iq := iq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus socks\_update';\ iq := iq'|\!\rangle$

(* **Summary:** receive RST and zap SYN_RECEIVED socket, removing from listen queue etc. *)

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{TCP}\ seg)) \wedge$

$(\exists seq\_discard\ ack\_discard\ URG\_discard\ ACK\_discard\ PSH\_discard\ SYN\_discard\ FIN\_discard\ win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard.$
$seg = \langle\!|$
$is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq\_discard : \mathrm{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack\_discard : \mathrm{tcp\_seq\_local});$
$URG := URG\_discard;$
$ACK := ACK\_discard;$
$PSH := PSH\_discard;$
$RST := \mathbf{T};$
$SYN := SYN\_discard;$
$FIN := FIN\_discard;$
$win := win\_discard;$
$ws := ws\_discard;$
$urp := urp\_discard;$
$mss := mss\_discard;$
$ts := ts\_discard;$
$data := data\_discard$
$|\!\rangle$
$) \wedge$

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

$sock = \mathrm{S{\scriptsize OCK}}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\mathrm{TCP\_Sock}(\mathrm{SYN\_RECEIVED}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

((* There is a corresponding listening socket – passive open *)
$(\exists (sid', lsock) :: socks \backslash\backslash sid.$
$\exists tcp\_lsock\ lis\ q0L\ q0R\ lsock'.$
$lsock.pr = \mathrm{TCP\_PROTO}(tcp\_lsock) \wedge$
$tcp\_lsock.st = \mathrm{LISTEN} \wedge$
$tcp\_lsock.lis = \uparrow lis \wedge$
$lis.q_0 = q0L\ @\ (sid :: q0R) \wedge$
$lsock' = lsock$
$\langle\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_lsock\ \langle\!|\ lis :=$
$\uparrow(lis\ \langle\!|\ q_0 := q0L\ @\ q0R|\!\rangle)|\!\rangle)|\!\rangle \wedge$
$socks\_update' = [(sid', lsock'); (sid, sock')]$
$) \vee$
((* No corresponding socket – simultaneous open *)
$socks\_update' = [(sid, sock')])) \wedge$

(* We do not delete the socket entry here because of simultaneous opens. Keep existing error for SYN_RECEIVED socket on RST *)
$sock' = (\text{tcp\_close}\ h.arch\ sock)\llparenthesis\ ps_1 := \mathbf{if}\ \ \text{bsd\_arch}\ h.arch\ \ \mathbf{then}\ \ *\ \ \mathbf{else}\ \ sock.ps_1\rrparenthesis$

deliver_in_7b **tcp: network nonurgent** **Receive RST and ignore for LISTEN socket**

$h\ \llparenthesis socks := socks \oplus [(sid, sock)];\ iq := iq\rrparenthesis \xrightarrow{\tau} h\ \llparenthesis socks := socks \oplus [(sid, sock)];\ iq := iq'\rrparenthesis$

(* **Summary:** receive RST and ignore for LISTEN socket *)

$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP}\ seg)) \wedge$
$sock = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, cantsndmore, cantrcvmore,$
$\quad \text{TCP\_Sock}(\text{LISTEN}, cb, lis, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

(* BSD listen bug – since we can call listen() from any state, the peer IP/port may have been set *)
$((is_2 = * \wedge ps_2 = *) \vee$
$(\text{bsd\_arch}\ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

$i_1 \in \text{local\_ips}\ h.ifds \wedge$
$\mathbf{T}\wedge$ (* placeholder for not a link-layer multicast or broadcast *)
(* seems unlikely, since $i_1 \in \text{local\_ips}\ h.ifds$ *)
$\neg(\text{is\_broadormulticast}\ h.ifds\ i_1) \wedge$
$\neg(\text{is\_broadormulticast}\ h.ifds\ i_2) \wedge$
$(\mathbf{case}\ \ is_1\ \ \mathbf{of}$
$\quad \uparrow i1' \rightarrow i1' = i_1\ \|$
$\quad * \rightarrow \mathbf{T}) \wedge$

$(\exists seq\_discard\ ack\_discard\ URG\_discard\ ACK\_discard\ PSH\_discard\ SYN\_discard\ FIN\_discard$
$\quad win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard.$
$seq = \llparenthesis$
$\quad is_1 := \uparrow i_2;$
$\quad is_2 := \uparrow i_1;$
$\quad ps_1 := \uparrow p_2;$
$\quad ps_2 := \uparrow p_1;$
$\quad seq := tcp\_seq\_flip\_sense(seq\_discard : \text{tcp\_seq\_foreign});$
$\quad ack := tcp\_seq\_flip\_sense(ack\_discard : \text{tcp\_seq\_local});$
$\quad URG := URG\_discard;$
$\quad ACK := ACK\_discard;$
$\quad PSH := PSH\_discard;$
$\quad RST := \mathbf{T};$
$\quad SYN := SYN\_discard;$
$\quad FIN := FIN\_discard;$
$\quad win := win\_discard;$
$\quad ws := ws\_discard;$
$\quad urp := urp\_discard;$
$\quad mss := mss\_discard;$
$\quad ts := ts\_discard;$
$\quad data := data\_discard$
$\quad \rrparenthesis$
$) \wedge$

$\text{tcp\_socket\_best\_match}(socks \backslash\backslash sid)(sid, sock) seg\ h.arch$ (* there does not exist a better socket match to which the segment should be sent *) *deliver_in_7c* **tcp: network nonurgent Receive RST and ignore for SYN_SENT(unacceptable ack) or TIME_WAIT socket**

$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ iq := iq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock')];\ iq := iq'|\!\rangle$

(* **Summary:** receive RST and ignore for SYN_SENT(unacceptable ack) or TIME_WAIT socket *)

$$
\begin{array}{l}
\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge \\
sid \notin \mathbf{dom}(socks) \wedge \\
sock = \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \\
\qquad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge \\
st \in \{\text{SYN\_SENT}; \text{TIME\_WAIT}\} \wedge \\
\\
(\exists seq\_discard\ URG\_discard\ PSH\_discard\ SYN\_discard\ FIN\_discard \\
\quad win\_discard\ ws\_discard\ urp\_discard\ mss\_discard\ ts\_discard\ data\_discard. \\
seg = \langle\!| \\
\qquad is_1 := \uparrow i_2; \\
\qquad is_2 := \uparrow i_1; \\
\qquad ps_1 := \uparrow p_2; \\
\qquad ps_2 := \uparrow p_1; \\
\qquad seq := tcp\_seq\_flip\_sense(seq\_discard : \text{tcp\_seq\_foreign}); \\
\qquad ack := tcp\_seq\_flip\_sense(ack : \text{tcp\_seq\_local}); \\
\qquad URG := URG\_discard; \\
\qquad ACK := ACK; \\
\qquad PSH := PSH\_discard; \\
\qquad RST := \mathbf{T}; \\
\qquad SYN := SYN\_discard; \\
\qquad FIN := FIN\_discard; \\
\qquad win := win\_discard; \\
\qquad ws := ws\_discard; \\
\qquad urp := urp\_discard; \\
\qquad mss := mss\_discard; \\
\qquad ts := ts\_discard; \\
\qquad data := data\_discard \\
\quad |\!\rangle \\
) \wedge
\end{array}
$$

(* no- or unacceptable- ACK *)

$$
\begin{array}{l}
(st = \text{SYN\_SENT} \implies \\
\quad (\neg ACK \vee (ACK \wedge \neg(cb.iss < ack \wedge ack \leq cb.snd\_max)))) \wedge \\
\\
sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge
\end{array}
$$

$(\mathbf{if}\ st = \text{TIME\_WAIT}\ \mathbf{then}$ (* only update if $\geq$ ESTABLISHED, c.f. `tcp\_input.c:887` *)

$$
\begin{array}{l}
\quad sock' = sock\ \langle\!|\ pr := \text{TCP\_PROTO}(tcp\_sock \\
\qquad\qquad \langle\!|\ cb := cb \\
\qquad\qquad\quad \langle\!|\ t\_idletime := \text{stopwatch\_zero};\ (\text{* just received segment *}) \\
\qquad\qquad\qquad tt\_keep := \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_IDLE}})|\!\rangle \\
\qquad\qquad |\!\rangle)|\!\rangle
\end{array}
$$

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 *)

$$
\begin{array}{l}
\mathbf{let}\ rcv\_window = \text{calculate\_bsd\_rcv\_wnd}\ sf\ tcp\_sock\ \mathbf{in} \\
sock' = sock\ \langle\!|\ pr := \text{TCP\_PROTO}(tcp\_sock \\
\qquad\qquad \langle\!|\ cb := cb
\end{array}
$$

$\quad\quad\quad (\!| \; rcv\_wnd := \textbf{if} \;\; \text{bsd\_arch} \;\; h.arch \;\; \textbf{then} \;\; rcv\_window \;\; \textbf{else} \;\; tcp\_sock.cb.rcv\_wnd;$
$\quad\quad\quad\;\; rcv\_adv := \textbf{if} \;\; \text{bsd\_arch} \;\; h.arch \;\; \textbf{then} \;\; tcp\_sock.cb.rcv\_nxt + rcv\_window$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else} \;\; tcp\_sock.cb.rcv\_adv$
$\quad\quad\quad |\!)$
$\quad\quad |\!))|\!))$

---

deliver_in_7d **tcp: network nonurgent** **Receive RST and zap SYN_SENT(acceptable ack) socket**

$h \;\langle\!|socks := socks \oplus [(sid, sock)];$ $\xrightarrow{\tau}$ $h \;\langle\!|socks := socks \oplus [(sid, sock')];$
$iq := iq|\!\rangle$ $\quad\quad iq := iq'|\!\rangle$

(* **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. *)

$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$
$sid \;\notin \textbf{dom}(socks) \wedge$
$sock = \text{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad\quad \text{TCP\_Sock}(\text{SYN\_SENT}, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge$

$(\exists seq\_discard \; URG\_discard \; PSH\_discard \; SYN\_discard \; FIN\_discard$
$\quad win\_discard \; ws\_discard \; urp\_discard \; mss\_discard \; ts\_discard \; data\_discard.$
$seg = \langle\!|$
$\quad is_1 := \uparrow i_2;$
$\quad is_2 := \uparrow i_1;$
$\quad ps_1 := \uparrow p_2;$
$\quad ps_2 := \uparrow p_1;$
$\quad seq := tcp\_seq\_flip\_sense(seq\_discard : \text{tcp\_seq\_foreign});$
$\quad ack := tcp\_seq\_flip\_sense(ack : \text{tcp\_seq\_local});$
$\quad URG := URG\_discard;$
$\quad ACK := \mathbf{T};$
$\quad PSH := PSH\_discard;$
$\quad RST := \mathbf{T};$
$\quad SYN := SYN\_discard;$
$\quad FIN := FIN\_discard;$
$\quad win := win\_discard;$
$\quad ws := ws\_discard;$
$\quad urp := urp\_discard;$
$\quad mss := mss\_discard;$
$\quad ts := ts\_discard;$
$\quad data := data\_discard$
$\quad |\!\rangle$
$) \wedge$

$cb.iss < ack \wedge ack \leq cb.snd\_max \wedge$ (* acceptable ack *)

$(\textbf{if} \;\; \text{windows\_arch} \;\; h.arch \;\; \textbf{then}$
$\quad sock' = sock$ (* Windows XP just ignores RST's with a valid ack during connection establishment *)
$\textbf{else}$
$\quad (\exists err.$
$\quad err \;\in \{\text{ECONNREFUSED}; \text{ECONNRESET}\} \wedge$ (* Note it is unclear whether or not this error will overwrite any existing error on the socket *)
$\quad sock' = (\text{tcp\_close} \;\; h.arch \;\; sock)(\!| \; ps_1 := \textbf{if} \;\; \text{bsd\_arch} \;\; h.arch \;\; \textbf{then} \;\; * \;\; \textbf{else} \;\; sock.ps_1;$
$\quad\quad\quad es := \uparrow err|\!)))$

---

deliver_in_8 **tcp: network nonurgent** **Receive SYN in non-{CLOSED; LISTEN; SYN_SENT; TIME_WAIT} state**

$$h\ \langle\!\langle socks := socks \oplus [(sid, sock)]; \quad \xrightarrow{\tau} \quad h\ \langle\!\langle socks := socks \oplus [(sid, sock')];$$
$$iq := iq; \qquad iq := iq';$$
$$oq := oq; \qquad oq := oq';$$
$$bndlm := bndlm\rangle\!\rangle \qquad bndlm := bndlm'\rangle\!\rangle$$

(* **Summary:** Receive a SYN in non-{CLOSED; LISTEN; SYN_SENT; TIME_WAIT} state. Drop it and (depending on the architecture) generate a RST. *)

$$
\begin{array}{l}
\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge \\
sid \notin \mathbf{dom}(socks) \wedge \\
sock = \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore, \\
\qquad \text{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)) \wedge \\
(\exists ws\_discard\ mss\_discard. \\
seg = \langle\!\langle \\
\quad is_1 := \uparrow i_2; \\
\quad is_2 := \uparrow i_1; \\
\quad ps_1 := \uparrow p_2; \\
\quad ps_2 := \uparrow p_1; \\
\quad seq := tcp\_seq\_flip\_sense(seq : \text{tcp\_seq\_foreign}); \\
\quad ack := tcp\_seq\_flip\_sense(ack : \text{tcp\_seq\_local}); \\
\quad URG := URG; \\
\quad ACK := ACK; \\
\quad PSH := PSH; \\
\quad RST := \mathbf{F}; \\
\quad SYN := \mathbf{T}; \\
\quad FIN := FIN; \\
\quad win := win; \\
\quad ws := ws\_discard; \\
\quad urp := urp; \\
\quad mss := mss\_discard; \\
\quad ts := ts; \\
\quad data := data \\
\quad \rangle\!\rangle \\
) \wedge
\end{array}
$$

(* Note that it may be the case that this rule should only apply when the SYN is *in the trimmed window*, should not it?; it's OK if there's a SYN bit set, for example in a retransmission. *)

$$st \notin \{\text{CLOSED}; \text{LISTEN}; \text{SYN\_SENT}; \text{TIME\_WAIT}\} \wedge$$

$$
\begin{array}{l}
sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge \\
\mathbf{let}\ \ t\_idletime' = \text{stopwatch\_zero}\ \ \mathbf{in} \\
\mathbf{let}\ \ tt\_keep' = \mathbf{if}\ \ tcp\_sock.st \neq \text{SYN\_RECEIVED}\ \ \mathbf{then} \\
\qquad\qquad \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_IDLE}}) \\
\qquad \mathbf{else} \\
\qquad\qquad tcp\_sock.cb.tt\_keep\ \ \mathbf{in} \\
\mathbf{let}\ \ tt\_fin\_wait\_2' = \mathbf{if}\ \ tcp\_sock.st = \text{FIN\_WAIT\_2}\ \ \mathbf{then} \\
\qquad\qquad \uparrow((())_{\text{slow\_timer TCPTV\_MAXIDLE}}) \\
\qquad \mathbf{else} \\
\qquad\qquad tcp\_sock.cb.tt\_fin\_wait\_2\ \ \mathbf{in}
\end{array}
$$

$$
\begin{array}{l}
sock' = sock\ \langle\!\langle\ pr := \text{TCP\_PROTO}(tcp\_sock \\
\qquad \langle\!\langle\ cb := tcp\_sock.cb\ \langle\!\langle\ tt\_keep := tt\_keep';
\end{array}
$$

$tt\_fin\_wait\_2 := tt\_fin\_wait\_2'$;
$t\_idletime := t\_idletime'$ $\rrbracket$
$\rrbracket)\rrbracket \wedge$

(**if** bsd_arch *h.arch* **then** make_rst_segment_from_cb $tcp\_sock.cb(i_1, i_2, p_1, p_2) seg'$ **else** **T**) $\wedge$
dropwithreset *seg h.ifds*(ticks_of *h.ticks*)BANDLIM_UNLIMITED *bndlm bndlm′ outsegs* $\wedge$
$outsegs' =$ (**if** bsd_arch *h.arch* **then** (TCP(seg')) :: *outsegs* **else** *outsegs*) $\wedge$
enqueue_each_and_ignore_fail *h.arch h.rttab h.ifds outsegs′ oq oq′*

deliver_in_9 **tcp: network nonurgent** **Receive SYN in TIME_WAIT state if there is no matching LISTEN socket or sequence number has not increased**

$h \ \llparenthesis socks := socks \oplus [(sid, sock)]; \quad \xrightarrow{\tau} \quad h \ \llparenthesis socks := socks \oplus [(sid, sock)];$
$iq := iq; \quad iq := iq';$
$oq := oq; \quad oq := oq';$
$bndlm := bndlm\rrparenthesis \quad bndlm := bndlm'\rrparenthesis$

(* **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*)) $\wedge$

$sid \notin$ **dom**(*socks*) $\wedge$
$sock =$ SOCK($\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
TCP_Sock(TIME_WAIT, $cb, *, sndq, sndurp, rcvq, rcvurp, iobc$)) $\wedge$

($\exists ws\_discard\ mss\_discard.$
$seg = \llparenthesis$
$is_1 := \uparrow i_2$;
$is_2 := \uparrow i_1$;
$ps_1 := \uparrow p_2$;
$ps_2 := \uparrow p_1$;
$seq := tcp\_seq\_flip\_sense(seq :$ tcp_seq_foreign);
$ack := tcp\_seq\_flip\_sense(ack :$ tcp_seq_local);
$URG := URG$;
$ACK := ACK$;
$PSH := PSH$;
$RST :=$ **F**;
$SYN :=$ **T**;
$FIN := FIN$;
$win := win$;
$ws := ws\_discard$;
$urp := urp$;
$mss := mss\_discard$;
$ts := ts$;
$data := data$
$\rrparenthesis$
) $\wedge$

(* no matching LISTEN socket, or the sequence number has not increased *)
(($seq \leq$ (tcp_sock_of *sock*)$.cb.rcv\_nxt$)
$\vee$
$\neg(\exists((sid, sock) :: socks) tcp\_sock.$
$sock.pr =$ TCP_PROTO(tcp_sock) $\wedge$
$tcp\_sock.st =$ LISTEN $\wedge$

$sock.is_1 \in \{*; \uparrow i_1\} \wedge$
$sock.ps_1 = \uparrow p_1)$
$) \wedge$

$(\textbf{if}\ \ \text{bsd\_arch}\ \ h.arch\ \ \textbf{then}\ \ \text{make\_rst\_segment\_from\_cb}\ \ cb(i_1, i_2, p_1, p_2) seg'\ \ \textbf{else}\ \ \mathbf{T}) \wedge$
$\text{dropwithreset}\ \ seg\ h.ifds(\text{ticks\_of}\ \ h.ticks)\text{BANDLIM\_RST\_CLOSEDPORT}\ bndlm\ bndlm'\ outsegs \wedge$
$outsegs' = (\textbf{if}\ \ \text{bsd\_arch}\ \ h.arch\ \ \textbf{then}\ (\text{TCP}(seg')) :: outsegs\ \ \textbf{else}\ \ outsegs) \wedge$
$\text{enqueue\_each\_and\_ignore\_fail}\ \ h.arch\ h.rttab\ h.ifds\ outsegs'\ oq\ oq'$

(* This rule does not appear in the BSD code; what happens there is that the old TIME_WAIT state socket is closed, and then the code jumps back to the top. So this rule covers the case where it then discovers nothing else is listening, like *deliver_in_5*. *)

Chapter 17

Host LTS: TCP Output

17.1 Output (TCP only)

A TCP implementation would typically perform output deterministically, e.g. during the processing a received segment it may construct and enqueue an acknowledgement segment to be emitted. This means that the detailed behaviour of a particular implementation depends on exactly where the output routines are called, affecting when segments are emitted. The contents of an emitted segment, on the other hand, must usually be determined by the socket state (especially the tcpcb), not from transient program variables, so that retransmissions can be performed.

In this specification we choose to be somewhat nondeterministic, loosely specifying when common-case TCP output to occur. This simplifies the modelling of existing implementations (avoiding the need to capture the code points at which the output routines are called) and should mean the specification is closer to capturing the set of all reasonable implementations.

A significant defect in the current specification is that it does not impose a very tight lower bound on how often output takes place. The satisfactory dynamic behaviour of TCP connections depends on an "ACK clock" property, with receivers acknowledging data sufficiently often to update the sender's send window. Characterising this may need additional constraints.

The rule presented in this chapter describes TCP output in the common case, i.e. the behaviour of TCP when emitting a non-SYN, non-RST segment. The whole behaviour is captured by the single rule *deliver_out_1* which relies upon the auxiliary functions tcp_output_required (p111) and tcp_output_really (p113). Output (strictly, adding segments to the host's output queue) may take place whenever this rule can fire; it does construct the output segments purely from the socket state.

The two auxiliary functions are loosely based on BSD's TCP output function, which can be logically divided into two halves. The first of these —to some approximation— is a guard that prevents output from 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_segment (p106) and make_syn_ack_segment (p107) and are output deterministically in response to either user events or segment input. SYN segments are emitted by the rules commonly involved in connection establishment, namely *connect_1*, *deliver_in_1*, *deliver_in_2*, *timer_tt_rexmtsyn_1* and *timer_tt_rexmt_1* and are special-cased in this way for clarity because connection establishment performs extra work such as option negotiation and state initialisation.

The creation of RST segments is performed by the auxiliaries make_rst_segment_from_cb (p109) and make_rst_segment_from_seg (p110), and are used by the rules that require a reset segment to be emitted in response to a user event, e.g. a close() call on a socket with a zero linger time, or as a socket's response to receiving some types of invalid segment.

In a few places, mainly in the specification of certain congestion control methods, some rules use tcp_output_really (p113) or the wrapper functions tcp_output_perhaps (p116) and mlift_tcp_output_perhaps_or_fail (p118) directly and—more importantly—deterministically. This is partly for clarity, perhaps because an RFC states that output "MUST" occur at that point, and partly for convenience, possibly because the model would require much extra state (hence adding unnecessary complexity) if the output function was not used in-place.

The tcp_output_perhaps function almost entirely mimics an implementation's TCP output function. It calls tcp_output_required to check that output can take place, applying any side-effects that it returns, and finally creates the segment with tcp_output_really. See tcp_output_perhaps (p116) and mlift_tcp_output_perhaps_or_fail (p118) for more information.

Other auxiliary functions are involved in TCP output and are described earlier. Once a segment has been constructed it is added to the host's output queue by one of enqueue_or_fail (p118), enqueue_or_fail_sock (p118), enqueue_and_ignore_fail (p118), enqueue_each_and_ignore_fail (p118) or mlift_tcp_output_perhaps_or_fail (p118). These functions are used by *deliver_out_1* and other rules in the specification to non-deterministically add a segment to the host's output queue. In the common case, a segment is added to the host's output queue successfully. In other cases, the auxiliary function rollback_tcp_output (p117) may assert a segment is unroutable and prevent the segment from being added to the queue. Some failures are non-deterministic in order to model "out of resource" style errors, although most are deterministic routing failures determined from the socket and host states. rollback_tcp_output has a second task to "undo" several of the socket's control block changes upon an error condition. Some of the enqueue functions ignore failure, e.g. enqueue_and_ignore_fail, and upon an error they just fail to queue the segment and do not update the socket with the "rolled-back" control block returned by rollback_tcp_output.

17.1.1 Summary

deliver_out_1	**tcp: network nonurgent**	Common case TCP output

17.1.2 Rules

deliver_out_1 **tcp: network nonurgent** **Common case TCP output**

$$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ oq := oq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock'')];\ oq := oq'|\!\rangle$$

(* **Summary:** output TCP segment if possible. In some cases update the socket's persist timer without performing output. *)

(* The TCP socket is connected *)
$sid \notin \mathbf{dom}(socks) \wedge$
$sock = \textsc{Sock}(fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore,$
$\quad cantrcvmore, \mathrm{TCP\_PROTO}(tcp\_sock)) \wedge$
$tcp\_sock = \mathrm{TCP\_Sock0}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc) \wedge$

(* and either is in a synchronised state with initial *SYN* acknowledged. . . *)
$((st \in \{\mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}; \mathrm{FIN\_WAIT\_1}; \mathrm{FIN\_WAIT\_2}; \mathrm{CLOSING};$
$\quad \mathrm{LAST\_ACK}; \mathrm{TIME\_WAIT}\} \wedge$
$\ cb.snd\_una \neq cb.iss) \vee$
(* . . . or is in the SYN_SENT or SYN_RECEIVED state and a *FIN* needs to be emitted *)
$(st \in \{\mathrm{SYN\_SENT}; \mathrm{SYN\_RECEIVED}\} \wedge cantsndmore \wedge cb.tf\_shouldacknow)$
$) \wedge$

(* A segment will be emitted if tcp_output_required asserts that a segment can be output (*do_output*). If tcp_output_required returns a function to alter the socket's persist timer (*persist_fun*), then this does not of itself mean that a segment is required, however *deliver_out_1* should still fire to allow the update to take place. *)
let $(do\_output, persist\_fun) = \mathrm{tcp\_output\_required}\ \ h.arch\ h.ifds\ sock$ **in**
$(do\_output \vee persist\_fun \neq *) \wedge$

(* Apply any persist timer side-effect from tcp_output_required *)

$\textbf{let}\ \ sock_0 = \textbf{option\_case}\ sock(\lambda f.sock$
$\quad\quad (\!| \ pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \ cb\ :\hat{=}\ f) |\!))persist\_fun\ \ \textbf{in}$

$(\textbf{if}\ \ do\_output\ \ \textbf{then}$ (* output a segment *)
(* Construct the segment to emit, updating the socket's state *)
$\mathrm{tcp\_output\_really}\ \ h.arch\ \mathbf{F}(\mathrm{ticks\_of}\ \ h.ticks)h.ifds\ sock_0(sock', outsegs') \wedge$

$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$

(* Add the segment to the host's output queue, rolling back the socket's control block state if an error occurs *)
$\mathrm{enqueue\_or\_fail\_sock}(tcp\_sock'.st\ \in \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\})h.arch\ h.rttab\ h.ifds$
$\quad outsegs'\ oq\ sock_0\ sock'(sock'', oq')$

$\textbf{else}$ (* Do not output a segment, but ensure things are tidied up *)
$oq = oq' \wedge$
$sock'' = sock_0$
)

Chapter 18

Host LTS: TCP Timers

18.1 Timers (TCP only)

18.1.1 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

18.1.2 Rules

timer_tt_rexmtsyn_1 **tcp: misc nonurgent** **SYN retransmit timer expires**

$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ oq := oq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock')];\ oq := oq'|\!\rangle$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.cb.tt\_rexmt = \uparrow(((\textsc{RexmtSyn}, shift))_d) \wedge$
$\mathrm{timer\_expires}\ d \wedge$ (* timer has expired *)
$tcp\_sock.st = \mathrm{SYN\_SENT} \wedge$ (* this rule is incomplete: RexmtSyn is possible in other states, since *deliver_in_2* may change state without clearing *tt_rexmt* *)

$cb = tcp\_sock.cb \wedge$

(**if** $shift + 1 \geq \mathrm{TCP\_MAXRXTSHIFT}$ **then**
(* Timer has expired too many times. Drop and close the connection *)

(* since socket state is SYN_SENT, no segments can be output *)
$\mathrm{tcp\_drop\_and\_close}\ h.arch(\uparrow \mathrm{ETIMEDOUT})sock(sock', []) \wedge$
$oq' = oq$

else
(* Update the control block based upon the number of occasions on which the timer expired *)

(**if** $shift + 1 = 1 \wedge cb.t\_rttinf.tf\_srtt\_valid$ **then** (* On the first retransmit store values for recovery from a bad retransmit *)
(* we cannot guess the safe window for this if we do not know the RTT, hence the second condition *)

$snd\_cwnd\_prev' = cb.snd\_cwnd\ \wedge$
$snd\_ssthresh\_prev' = cb.snd\_ssthresh\ \wedge$
$t\_badrxtwin' = (())^{\text{TimeWindow}}_{\text{kern\_timer}(\text{time}(cb.t\_rttinf.t\_srtt/2))}$ (* kern_timer for a ticks-based deadline *)
else (* Otherwise keep the previous values *)
$snd\_cwnd\_prev' = cb.snd\_cwnd\_prev\ \wedge$
$snd\_ssthresh\_prev' = cb.snd\_ssthresh\_prev\ \wedge$
$t\_badrxtwin' = cb.t\_badrxtwin$ (* should be TimeWindowClosed, since retransmit timer is always longer than $t\_srtt/2$ *)
$)\ \wedge$

(**if** $(shift + 1 = 3) \wedge \neg(\text{linux\_arch}\ \ h.arch)$ **then** (* On the third retransmit turn off window scaling and timestamping options *)
$tf\_req\_tstmp' = \mathbf{F}\ \wedge$
$request\_r\_scale' = *$
else (* Otherwise keep the previous values *)
$tf\_req\_tstmp' = cb.tf\_req\_tstmp\ \wedge$
$request\_r\_scale' = cb.request\_r\_scale$
$)\ \wedge$

let $t\_rttinf' =$
(**if** $shift + 1 >$ TCP_MAXRXTSHIFT **div** 4 **then**
(* Invalidate the recorded smoothed round-trip time for the connection after TCP_MAXRXTSHIFT **div** 4 retransmits *)
(* Note that the BSD code adjusts the *srtt* and *rttvar* values here to ensure that if it does not get a new rtt measurement before the next retransmit it can still use the existing values. We do not need to do this for two reasons: (1) we have a flag to invalidate the *srtt* values (the only reason BSD updates *srtt* to be zero and hacks *rrttvar* is to mark it invalid and request a new rtt update), and (2) the BSD_RTTVAR_BUG does not affect SYN retransmits in any case (because for SYN retransmits *srtt* is zero and BSD hacks up *rttvar* appropriately at the start of a new connection to make everything just work) *)
(* Note that the socket's route should be discarded. *)
$cb.t\_rttinf\ \langle\!|\ tf\_srtt\_valid := \mathbf{F}|\!\rangle$
else
$cb.t\_rttinf$) **in**

$cb' = cb\ \langle\!|$ (* Restart the *rexmt* timer to time the retransmitted SYN *)
$tt\_rexmt := \text{start\_tt\_rexmtsyn}\ \ h.arch(shift + 1)\mathbf{F}\ cb.t\_rttinf;$
(* reset to next backoff point *)
$t\_badrxtwin := t\_badrxtwin';$
$t\_rttinf := t\_rttinf'$
$\langle\!|\ t\_lastshift := shift + 1;$
$t\_wassyn := \mathbf{T}|\!\rangle;$
$tf\_req\_tstmp := tf\_req\_tstmp';$
$request\_r\_scale := request\_r\_scale';$
$snd\_nxt := cb.iss + 1;$ (* value after sending SYN *)
$snd\_recover := cb.iss + 1;$ (* value after sending SYN *)

$t\_rttseg := *;$
$snd\_cwnd := cb.t\_maxseg;$
(* Calculation as per BSD *)
$snd\_ssthresh := cb.t\_maxseg * \mathbf{max}\ \ 2(\mathbf{min}\ \ cb.snd\_wnd\ cb.snd\_cwnd$
$\mathbf{div}(2 * cb.t\_maxseg));$
$snd\_cwnd\_prev := snd\_cwnd\_prev';$
$snd\_ssthresh\_prev := snd\_ssthresh\_prev';$
$t\_dupacks := 0|\!\rangle\ \wedge$

$(\exists i_1\ i_2\ p_1\ p_2.(sock.is_1, sock.is_2, sock.ps_1, sock.ps_2) = (\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2)\ \wedge$

(* Create the segment to be retransmitted *)
choose $seg' :: (\text{make\_syn\_segment}\ \ cb'(i_1, i_2, p_1, p_2)(\text{ticks\_of}\ \ h.ticks)).$

(* Attempt to add the new segment to the host's output queue, constraining the final control block state *)
enqueue_or_fail **F** $h.arch$ $h.rttab$ $h.ifds$[TCP seg']oq
$(cb\ \llparenthesis\ snd\_nxt := cb.iss;\ tt\_delack := *;$
$last\_ack\_sent := \mathrm{tcp\_seq\_foreign}\ 0w;\ rcv\_adv := \mathrm{tcp\_seq\_foreign}\ 0w$
$\rrparenthesis)\,cb'(cb'', oq')$
$) \wedge$
$sock' = sock\ \llparenthesis\ pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \llparenthesis\ cb := cb''\rrparenthesis)\rrparenthesis$
$)$

timer_tt_rexmt_1 **tcp: misc nonurgent** **retransmit timer expires**

$h\ \llparenthesis socks := socks \oplus [(sid, sock)];\ oq := oq\rrparenthesis \xrightarrow{\tau} h\ \llparenthesis socks := socks \oplus [(sid, sock'')];\ oq := oq'\rrparenthesis$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$
$(tcp\_sock.st \notin \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}; \mathrm{CLOSE\_WAIT}; \mathrm{FIN\_WAIT\_2}; \mathrm{TIME\_WAIT}\} \vee$
$(tcp\_sock.st = \mathrm{LISTEN} \wedge \mathrm{bsd\_arch}\ \ h.arch)) \wedge$

$tcp\_sock.cb.tt\_rexmt = \uparrow(((\mathrm{Rexmt}, shift))_d) \wedge$
timer_expires $d \wedge$

$cb = tcp\_sock.cb \wedge$

(**if** $shift + 1 >$ (**if** $tcp\_sock.st = \mathrm{SYN\_RECEIVED}$ **then** TCP_SYNACKMAXRXTSHIFT
else TCP_MAXRXTSHIFT) **then**
(* Note that BSD's syncaches have a much lower threshold for retransmitting SYN,ACKs than normal *)
(* drop connection *)

tcp_drop_and_close $h.arch(\uparrow \mathrm{ETIMEDOUT})sock(sock', [\mathrm{TCP}\ seg'])$ (* will always get exactly one segment *)

else

(* on first retransmit, store values for recovery from bad retransmit *)
(* we cannot guess the safe window for this if we do not know the RTT, hence the second condition *)
(**if** $shift + 1 = 1 \wedge cb.t\_rttinf.tf\_srtt\_valid$ **then**
$snd\_cwnd\_prev' = cb.snd\_cwnd \wedge$
$snd\_ssthresh\_prev' = cb.snd\_ssthresh \wedge$
$t\_badrxtwin' = (())^{\mathrm{TimeWindow}}_{\mathrm{kern\_timer}(\mathsf{time}(cb.t\_rttinf.t\_srtt/2))}$ (* kern_timer for a ticks-based deadline *)
else
$snd\_cwnd\_prev' = cb.snd\_cwnd\_prev \wedge$
$snd\_ssthresh\_prev' = cb.snd\_ssthresh\_prev \wedge$
$t\_badrxtwin' = cb.t\_badrxtwin) \wedge$ (* should be TimeWindowClosed, since retransmit timer is always longer than $t\_srtt/2$ *)

(* NB: The socket is not in SYN_SENT here; the rexmt timer has been split into two, and SYN_SENT uses *tt_rexmtsyn*. *)

let $t\_rttinf' =$ (**if** $shift + 1 >$ TCP_MAXRXTSHIFT **div** 4 **then**
(* Note that the socket's route should be discarded. *)
$cb.t\_rttinf\ \llparenthesis$
$tf\_srtt\_valid := \mathbf{F};$
$t\_srtt\ :\hat{=} (cb.t\_rttinf.t\_srtt/4)$
onlywhen(bsd_arch $h.arch \wedge$ BSD_RTTVAR_BUG)

$\rrbracket$
else
$cb.t\_rttinf$) **in**

(* backoff the timer and do a retransmit *)
$cb' = cb$ $\llbracket$ $tt\_rexmt :=$ start_tt_rexmt $h.arch(shift + 1)$**F** $cb.t\_rttinf$; (* reset to next backoff point *)
(* tcp_output_really touches this again, but actually leaves it the same, unless $sock.snd\_urp$ is set and $win_0 \neq 0$, weirdly *)
$t\_badrxtwin := t\_badrxtwin'$;
$t\_rttinf := t\_rttinf'$ $\llbracket$
$t\_lastshift := shift + 1$;
$t\_wassyn :=$ **F**
$\rrbracket$;
$snd\_nxt := cb.snd\_una$; (* want to retransmit from snd_una *)
$snd\_recover := cb.snd\_max$;
$t\_rttseg := *$;
$snd\_cwnd := cb.t\_maxseg$;
$snd\_ssthresh := cb.t\_maxseg * \mathbf{max}\ 2(\mathbf{min}\ cb.snd\_wnd\ cb.snd\_cwnd\ \mathbf{div}(2 * cb.t\_maxseg))$;
$snd\_cwnd\_prev := snd\_cwnd\_prev'$;
$snd\_ssthresh\_prev := snd\_ssthresh\_prev'$;
$t\_dupacks := 0\rrbracket \wedge$

(**if** $tcp\_sock.st =$ 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 $cb''' =$
(**if** ((linux_arch $h.arch) \wedge cb.tf\_req\_tstmp$) **then**
cb' $\llbracket$ $tf\_req\_tstmp :=$ **T**;
$tf\_doing\_tstmp :=$ **T**$\rrbracket$
else
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. *)
$seg' \in$ make_syn_ack_segment $cb'''(i_1, i_2, p_1, p_2)$(ticks_of $h.ticks) \wedge$

(* 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 *)
$sock' = sock$ $\llbracket$ $pr :=$ TCP_PROTO(tcp_sock $\llbracket$ $cb := cb'$
$\llbracket$ $snd\_nxt := cb'.snd\_nxt + 1\rrbracket\rrbracket)\rrbracket$
)

else if $tcp\_sock.st = \mathrm{LISTEN}$ **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. *)

$\mathrm{bsd\_arch}\ \ h.arch\ \wedge$
$(\exists i_1\ i_2\ p_1\ p_2.$
$(sock.is_1, sock.is_2, sock.ps_1, sock.ps_2) = (\uparrow i_1, \uparrow i_2, \uparrow p_1, \uparrow p_2)\ \wedge$
$seg' \ \in\ \mathrm{bsd\_make\_phantom\_segment}\ \ cb'(i_1, i_2, p_1, p_2)(\mathrm{ticks\_of}\ \ h.ticks)(sock.cantsndmore))\ \wedge$
(* Retransmission only continues if *FIN* is set in the outgoing segment (really!) *)
$sock' = sock\ \langle\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock$
$\quad\langle\!|\ cb := cb'\ \langle\!|\ tt\_rexmt\ :\hat{=}\ *\ \textbf{onlywhen}\,\neg seg'.FIN|\!\rangle|\!\rangle)|\!\rangle$

else (* ESTABLISHED, FIN_WAIT_1, CLOSING, LAST_ACK *)
(* i.e., cannot be CLOSED, LISTEN, SYN_SENT, CLOSE_WAIT, FIN_WAIT_2, TIME_WAIT *)
$\mathrm{tcp\_output\_really}\ \ h.arch\ \mathbf{F}(\mathrm{ticks\_of}\ \ h.ticks)h.ifds$
$\quad(sock\ \langle\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \langle\!|\ cb := cb'|\!\rangle)|\!\rangle)$
$\quad(sock', [\mathrm{TCP}\ \ seg'])$ (* always emits exactly one segment *)

)

$)\ \wedge$

$\mathrm{enqueue\_or\_fail}\ \ \mathbf{T}\ h.arch\ h.rttab\ h.ifds[\mathrm{TCP}\ \ seg']oq$
$\quad cb'\ tcp\_sock'.cb(cb'', oq')\ \wedge$
$sock'' = sock'\ \langle\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock'\ \langle\!|\ cb := cb''|\!\rangle)|\!\rangle$

---

timer_tt_persist_1 **tcp: misc nonurgent** **persist timer expires**

$h\ \langle\!|socks := socks \oplus [(sid, sock)]; oq := oq|\!\rangle \ \xrightarrow{\tau}\ h\ \langle\!|socks := socks \oplus [(sid, sock'')]; oq := oq'|\!\rangle$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock)\ \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock')\ \wedge$
$tcp\_sock.cb.tt\_rexmt = \uparrow(((\mathrm{P{\scriptsize ERSIST}}, shift))_d)\ \wedge$
$\mathrm{timer\_expires}\ \ d\ \wedge$
$\textbf{let}\ \ sock_0 = sock\ \langle\!|\ pr := \mathrm{TCP\_PROTO}(tcp\_sock$
$\quad\langle\!|\ cb := tcp\_sock.cb$
$\quad\quad\langle\!|\ tt\_rexmt := \mathrm{start\_tt\_persist}(shift + 1)tcp\_sock.cb.t\_rttinf\ \ h.arch|\!\rangle|\!\rangle)|\!\rangle\ \textbf{in}$

$\mathrm{tcp\_output\_really}\ \ h.arch\ \mathbf{T}$ (* **T** indicates a window probe is requested *)
$\quad(\mathrm{ticks\_of}\ \ h.ticks)h.ifds$
$\quad sock_0$
$\quad(sock', outsegs')\ \wedge$
$\mathrm{enqueue\_or\_fail\_sock}(tcp\_sock'.st\ \in \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\})h.arch\ h.rttab\ h.ifds$
$\quad outsegs'\ oq\ sock_0\ sock'(sock'', oq')$

---

timer_tt_keep_1 **tcp: network nonurgent** **keepalive timer expires**

$h\ \langle\!|socks := socks \oplus$
$\quad[(sid, \mathrm{S{\scriptsize OCK}}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$
$\quad\quad\mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))];$
$oq := oq|\!\rangle$

$$\xrightarrow{\tau} \quad h \; \langle\!|socks := socks \oplus$$
$$[(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, es, cantsndmore, cantrcvmore,$$
$$\text{TCP\_Sock}(st, cb', *, sndq, sndurp, rcvq, rcvurp, iobc)))];$$
$$oq := oq' |\!\rangle$$

(* 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 $d \wedge$
(* Note the following condition also needs to be investigated: $cb.t\_rcvtime + tcp\_keepidle + tcp\_keepcnt * tcp\_keepintvl < NOW \wedge$ – still probing *)
$(\exists win\_.$
w2n $win\_ = cb.rcv\_wnd \gg cb.rcv\_scale \wedge$

let $ts =$ **if** $cb.tf\_doing\_tstmp$ **then**
  let $ts\_ecr' =$ **option_case** (ts_seq $0w$) **I** (timewindow_val_of $cb.ts\_recent$) **in**
  $\uparrow((\text{ticks\_of } h.ticks), ts\_ecr')$
 else
  $*$ **in**

$seg = \langle\!|\; is_1 := \uparrow i_2;$
$is_2 := \uparrow i_1;$
$ps_1 := \uparrow p_2;$
$ps_2 := \uparrow p_1;$
$seq := cb.snd\_una - 1;$ (* deliberately outside window *)
$ack := cb.rcv\_nxt;$
$URG := \mathbf{F};$
$ACK := \mathbf{T};$
$PSH := \mathbf{F};$
$RST := \mathbf{F};$
$SYN := \mathbf{F};$
$FIN := \mathbf{F};$
$win := win\_;$
$ws := *;$
$urp := 0w;$
$mss := *;$
$ts := ts;$
$data := [\,]$
$|\!\rangle) \wedge$

enqueue_and_ignore_fail $h.arch\ h.rttab\ h.ifds[\text{TCP } seg]\ oq\ oq' \wedge$
$cb' = cb\ \langle\!|\; tt\_keep := \uparrow((())_{\text{slow\_timer TCPTV\_KEEPINTVL}});$
$last\_ack\_sent := seg.ack$
$|\!\rangle$

timer_tt_2msl_1 **tcp: misc nonurgent** **2*MSL timer expires**

$$h\ \langle\!|socks := socks \oplus [(sid, sock)]|\!\rangle \quad \xrightarrow{\tau} \quad h\ \langle\!|socks := socks \oplus [(sid, sock')]|\!\rangle$$

(* **Summary:** When the 2MSL TIME_WAIT period expires, the socket is closed. *)

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.cb.tt\_2msl = \uparrow((())_d) \wedge$
$\mathrm{timer\_expires}\ d \wedge$
$sock' = \mathrm{tcp\_close}\ h.arch\ sock$

---

timer_tt_delack_1 **tcp: misc nonurgent** **delayed-ACK timer expires**

$$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ oq := oq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock'')];\ oq := oq'|\!\rangle$$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$sock'.pr = \mathrm{TCP\_PROTO}(tcp\_sock') \wedge$
$tcp\_sock.cb.tt\_delack = \uparrow((())_d) \wedge$
$\mathrm{timer\_expires}\ d \wedge$
$\mathbf{let}\ \ sock_0 = sock\ \langle\!| pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \langle\!| cb := tcp\_sock.cb\ \langle\!| tt\_delack := * |\!\rangle |\!\rangle) |\!\rangle\ \mathbf{in}$
$\mathrm{tcp\_output\_really}\ \ h.arch\ \mathbf{F}(\mathrm{ticks\_of}\ \ h.ticks) h.ifds\ sock_0 (sock', outsegs') \wedge$
$\mathrm{enqueue\_or\_fail\_sock}(tcp\_sock'.st\ \in \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\}) h.arch\ h.rttab\ h.ifds$
$\quad outsegs'\ oq\ sock_0\ sock' (sock'', oq')$

Description
This overlaps with *deliver_out_1*. This is a bit odd, but is a consequence of our liberal nondeterministic TCP output.

---

timer_tt_conn_est_1 **tcp: misc nonurgent** **connection establishment timer expires**

$$h\ \langle\!|socks := socks \oplus [(sid, sock)];\ oq := oq|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock')];\ oq := oq'|\!\rangle$$

(* **Summary:** If the connection-establishment timer goes off, drop the connection (possibly *RST*ing the other end). *)

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.cb.tt\_conn\_est = \uparrow((())_d) \wedge$
$\mathrm{timer\_expires}\ d \wedge$
$\mathrm{tcp\_drop\_and\_close}\ \ h.arch(\uparrow \mathrm{ETIMEDOUT})$
$\quad (sock\ \langle\!| pr := \mathrm{TCP\_PROTO}(tcp\_sock\ \langle\!| cb := tcp\_sock.cb$
$\quad\quad \langle\!| tt\_conn\_est := * |\!\rangle |\!\rangle) |\!\rangle)(sock', outsegs) \wedge$
(* Note it should be the case that the socket is in SYN_SENT, and so *outsegs* will be empty, but that is not definite. *)
$\mathrm{enqueue\_and\_ignore\_fail}\ \ h.arch\ h.rttab\ h.ifds\ outsegs\ oq\ oq'$

Description POSIX: says, in the *INFORMATIVE* section *APPLICATION USAGE*, that the state of the socket is unspecified if `connect()` fails. We could (in the POSIX "architecture") model this accurately.

---

timer_tt_fin_wait_2_1 **tcp: misc nonurgent** **FIN_WAIT_2 timer expires**

$$h\ \langle\!|socks := socks \oplus [(sid, sock)]|\!\rangle \xrightarrow{\tau} h\ \langle\!|socks := socks \oplus [(sid, sock')]|\!\rangle$$

$sock.pr = \mathrm{TCP\_PROTO}(tcp\_sock) \wedge$
$tcp\_sock.cb.tt\_fin\_wait\_2 = \uparrow((())_d) \wedge$

$\text{timer\_expires}\ d\ \wedge$
$sock' = \text{tcp\_close}\ h.arch\ sock$

Description This stops the timer and closes the socket.

Unlike BSD, we take steps to ensure that this timer only fires when it is really time to close the socket. Specifically, we reset it every time we receive a segment while in FIN_WAIT_2, to TCPTV_MAXIDLE. This means we do not need any guarding conditions here; we just do it.

This means that we do not directly model the BSD behaviour of "sleep for 10 minutes, then check every 75 seconds to see if the connection has been idle for 10 minutes".

Chapter 19

Host LTS: UDP Input Processing

19.1 Input Processing (UDP only)

19.1.1 Summary

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

19.1.2 Rules

deliver_in_udp_1 **udp: network nonurgent** **Get UDP datagram from host's in-queue and deliver it to a matching socket**

$$\begin{aligned} h_0 \xrightarrow{\tau} \; & h_0 \,⦇ iq := iq'; \\ & socks := socks \oplus \\ & \quad [(sid, sock \;\; pr := \mathrm{UDP\_Sock}(rcvq'))] ⦈ \end{aligned}$$

$h_0 = h$ ⦇ $iq := iq;$
$\quad socks := socks \oplus$
$\quad\quad [(sid, sock \;\; pr := \mathrm{UDP\_Sock}(rcvq))]$⦈ $\wedge$
$rcvq' = rcvq \mathbin{@} [\mathrm{D{\scriptsize GRAM\_MSG}}($⦇ $data := data; is := \uparrow i_3; ps := ps_3$⦈$)] \wedge$
$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{UDP}($⦇ $is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3; ps_2 := ps_4; data := data$⦈$))) \wedge$
$(\exists (ifid, \mathsf{ifd}) :: (h_0.ifds).i_4 \;\in\; \mathsf{ifd}.ipset) \wedge$
$sid \;\in\; \mathrm{lookup\_udp}\;\; h_0.socks(i_3, ps_3, i_4, ps_4) h_0.bound \;\; h_0.arch \wedge$
$\mathbf{T}\wedge$ (* placeholder for "not a link-layer multicast or broadcast" *)
$\neg(\mathrm{is\_broadormulticast}\;\; h_0.ifds\;\; i_4) \wedge$ (* seems unlikely, since $i_1 \;\in\; \mathrm{local\_ips}\;\; h.ifds$ *)
$\neg(\mathrm{is\_broadormulticast}\;\; h_0.ifds\;\; i_3)$

Description

At the head of the host's in-queue is a UDP datagram with source address $(\uparrow i_3, ps_3)$, destination address $(\uparrow i_4, ps_4)$, and data *data*. The destination IP address, i_4, is an IP address for one of the host's interfaces and is not an IP- or link-layer broadcast or multicast address and neither is the source IP address, i_3.

The UDP socket *sid* matches the address quad of the datagram (see lookup_udp (p86) for details). A τ transition is made. The datagram is removed from the host's in-queue, *iq*, and appended to the tail of the socket's receive queue, *rcvq'*, leaving the host with in-queue *iq'* and the socket with receive queue *rcvq'*.

deliver_in_udp_2 **udp: network nonurgent** **Get UDP datagram from host's in-queue but generate ICMP, as no matching socket**

$h\ \ iq := iq \xrightarrow{\tau} h\ \langle\!|iq := iq'; oq := \mathbf{if}\ \ icmp\_to\_go\ \ \mathbf{then}\ \ oq'\ \ \mathbf{else}\ \ h.oq|\!\rangle$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{UDP}(\langle\!|\ is_1 := \uparrow i_3; is_2 := \uparrow i_4; ps_1 := ps_3;$
$ps_2 := ps_4; data := data|\!\rangle))) \wedge$
$\mathrm{lookup\_udp}\ \ h.socks(i_3, ps_3, i_4, ps_4) h.bound\ h.arch = \emptyset \wedge$
$icmp = \mathrm{ICMP}(\langle\!|\ is_1 := \uparrow i_4; is_2 := \uparrow i_3; is_3 := \uparrow i_3; is_4 := \uparrow i_4;$
$ps_3 := ps_3; ps_4 := ps_4; proto := \mathrm{PROTO\_UDP}; seq := *;$
$t := \mathrm{ICMP\_UNREACH(PORT)}|\!\rangle) \wedge$
$(\mathrm{enqueue\_oq}(h.oq, icmp, oq', \mathbf{T}) \vee icmp\_to\_go = \mathbf{F})$ (* non-deterministic ICMP generation *) $\wedge$
$i_4\ \in\ \mathrm{local\_ips}\ \ h.ifds \wedge$
$\mathbf{T}\wedge$ (* placeholder for "not a link-layer multicast or broadcast" *)
$\neg(\mathrm{is\_broadormulticast}\ \ h.ifds\ i_4)\wedge$ (* seems unlikely, since $i_1\ \in\ \mathrm{local\_ips}\ \ h.ifds$ *)
$\neg(\mathrm{is\_broadormulticast}\ \ h.ifds\ i_3)$

Description

At the head of the host's in-queue, iq, is a UDP datagram with source address $(\uparrow i_3, ps_3)$, destination address $(\uparrow i_4, ps_4)$, and data $data$. The destination IP address, i_4, is an IP address for one of the host's interfaces and is neither a broadcast or multicast address; the source IP address, i_3, is also not a broadcast or multicast address. None of the sockets in the host's finite map of sockets, $h.socks$, match the datagram (see lookup_udp (p86) for details).

A τ transition is made. The datagram is removed from the host's in-queue, leaving it with in-queue iq'. An ICMP Port-unreachable message may be generated and appended to the tail of the host's out-queue in response to the datagram.

deliver_in_udp_3 **udp: network nonurgent** **Get UDP datagram from host's in-queue and drop as from a martian address**

$h\ \langle\!|iq := iq|\!\rangle \xrightarrow{\tau} h\ \langle\!|iq := iq'|\!\rangle$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{UDP}\ dgram)) \wedge$
$dgram.is_2 = \uparrow i_2 \wedge$
$is_1 = dgram.is_1 \wedge$
$i_2\ \in\ \mathrm{local\_ips}(h.ifds) \wedge$
$(\mathbf{F} \vee$
$\neg(\mathbf{T} \wedge$
$\neg(\mathrm{is\_broadormulticast}\ \ h.ifds\ i_2)\wedge$ (* seems unlikely, since $i_1\ \in\ \mathrm{local\_ips}\ \ h.ifds$ *)
$\neg(is_1 = *) \wedge$
$\neg\,\mathrm{is\_broadormulticast}\ \ h.ifds(\mathbf{the}\ \ is_1)$
$)$
$)$

Description

At the head of the host's in-queue, iq, is a UDP datagram with destination IP address $\uparrow i_2$ which is an IP address for one of the host's interfaces. Either i_2 is an IP-layer broadcast or multicast address, or the source IP address, is_1, is not set or is an IP-layer broadcast or multicast address.

A τ transition is made. The datagram is dropped from the host's in-queue, leaving it with in-queue iq'.

Chapter 20

Host LTS: ICMP Input Processing

20.1 Input Processing (ICMP only)

20.1.1 Summary

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

20.1.2 Rules

deliver_in_icmp_1 **all: network nonurgent** **Receive** *ICMP_UNREACH_NET* **etc for known socket**

$$h_0 \xrightarrow{\tau} h \,\langle\!|\, socks := socks \oplus [(sid, sock')]; \; iq := iq'; \; oq := oq' \,|\!\rangle$$

$h_0 = h \,\langle\!|\, socks := socks \oplus [(sid, sock)];$
$\quad iq := iq;$
$\quad oq := oq \,|\!\rangle \wedge$
$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{ICMP}\ icmp)) \wedge$
$icmp.t \in \{\mathrm{ICMP\_UNREACH}\ c \mid$
$\quad c \in \{\mathrm{NET}; \mathrm{HOST}; \mathrm{SRCFAIL}; \mathrm{NET\_UNKNOWN}; \mathrm{HOST\_UNKNOWN}; \mathrm{ISOLATED};$
$\quad\quad \mathrm{TOSNET}; \mathrm{TOSHOST}; \mathrm{PREC\_VIOLATION}; \mathrm{PREC\_CUTOFF}\}\} \wedge$
$icmp.is_3 = \uparrow i_3 \wedge$
$i_3 \notin \mathrm{IN\_MULTICAST} \wedge$
$sid \in \mathrm{lookup\_icmp}\ h_0.socks\ icmp\ h_0.arch\ h_0.bound \wedge$
$(\textbf{case}\ sock.pr\ \textbf{of}$
$\quad \mathrm{TCP\_PROTO}(tcp\_sock) \rightarrow$
$\quad\quad (\exists icmp_{seq}.icmp.seq = \uparrow icmp_{seq} \wedge$
$\quad\quad \textbf{if}\ tcp\_sock.cb.snd\_una \leq icmp_{seq} \wedge icmp_{seq} < tcp\_sock.cb.snd\_max\ \textbf{then}$
$\quad\quad\quad \textbf{if}\ tcp\_sock.st = \mathrm{ESTABLISHED}\ \textbf{then}$
$\quad\quad\quad\quad sock' = sock \wedge$ (* ignore transient error while connected *)
$\quad\quad\quad\quad oq' = oq$
$\quad\quad\quad \textbf{else if}\ tcp\_sock.st \in \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}; \mathrm{SYN\_RECEIVED}\} \wedge$

$tcp\_sock.cb.tt\_rexmt \neq * \wedge$ shift_of $tcp\_sock.cb.tt\_rexmt > 3 \wedge$
$tcp\_sock.cb.t\_softerror \neq *$ **then**
tcp_drop_and_close $h.arch(\uparrow$ EHOSTUNREACH$)sock(sock', outsegs) \wedge$
enqueue_and_ignore_fail $h.arch$ $h.rttab$ $h.ifds$ $outsegs$ oq oq'
else
$sock' = sock$ ⦇ $pr :=$ TCP_PROTO(tcp_sock
⦇ $cb := tcp\_sock.cb$
⦇ $t\_softerror := \uparrow$ EHOSTUNREACH⦈⦈)⦈ $\wedge$
$oq' = oq$
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
$sock' = sock \wedge$
$oq' = oq) \parallel$
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**

$h_0 \xrightarrow{\tau} h$ ⦇$socks := socks \oplus$
$[(sid, sock')]$;
$iq := iq'$;
$oq := oq'$⦈

$h_0 = h$ ⦇ $socks := socks \oplus$
$[(sid, sock)]$;
$iq := iq$;
$oq := oq$⦈ $\wedge$
dequeue_iq$(iq, iq', \uparrow($ICMP $icmp)) \wedge$
$icmp.t =$ ICMP_UNREACH(NEEDFRAG $icmpmtu$) $\wedge$
$(icmp.is_3 = * \vee$ **the** $icmp.is_3 \notin$ IN_MULTICAST$) \wedge$
$sid \in$ 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**
$\uparrow mtu \rightarrow$ **w2n** mtu
$\parallel * \rightarrow$ next_smaller(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**
TCP_PROTO(tcp_sock) $\rightarrow$

$(\exists icmp_{seq}.icmp.seq = \uparrow icmp_{seq} \wedge$
if **is_some** $icmp.is_3$ **then**
$($**if** $tcp\_sock.cb.snd\_una \leq icmp_{seq} \wedge icmp_{seq} < tcp\_sock.cb.snd\_max$ **then**
if $nextmtu = *$ **then**
$sock' = sock \,⦃\, pr := \mathrm{TCP\_PROTO}(tcp\_sock$
$⦃\, cb := tcp\_sock.cb \,⦃\, t\_maxseg := \mathrm{MSSDFLT}⦄⦄)⦄ \wedge$
$oq' = oq$
else
let $mss = \mathbf{min}(sock.sf.n(\mathrm{SO\_SNDBUF}))$
(rounddown MCLBYTES
$($**the** $nextmtu - 40 - ($**if** $tcp\_sock.cb.tf\_doing\_tstmp$ **then** 12 **else** $0)))$ **in**
(* BSD: TS, plus NOOP for alignment *)
if $mss \leq tcp\_sock.cb.t\_maxseg$ **then**
let $sock'' = sock \,⦃\, pr := \mathrm{TCP\_PROTO}(tcp\_sock$
$⦃\, cb := tcp\_sock.cb$
$⦃\, t\_maxseg := mss;$
$t\_rttseg := *;$
$snd\_nxt := tcp\_sock.cb.snd\_una$
$⦄⦄)⦄$ **in**
$\exists sock'''\ outsegs\ tcp\_sock'''.$
$sock'''.pr = \mathrm{TCP\_PROTO}(tcp\_sock''') \wedge$
$\mathrm{tcp\_output\_perhaps}\ h.arch(\mathrm{ticks\_of}\ \ h.ticks)h.ifds\ sock''(sock''', outsegs) \wedge$
$\mathrm{enqueue\_or\_fail\_sock}(tcp\_sock'''.st \notin \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\})$
$h.arch\ h.rttab\ h.ifds\ outsegs\ oq$
$sock''\ sock'''(sock', oq')$
else
$sock' = sock \wedge oq' = oq$
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
$sock' = sock \wedge oq' = oq)$
else
$sock' = sock \wedge oq' = oq) \,\|$
$\mathrm{UDP\_PROTO}(udp\_sock) \rightarrow$
if $\mathrm{windows\_arch}\ h.arch$ **then**
$sock' = sock \,⦃\, pr := \mathrm{UDP\_PROTO}(udp\_sock$
$⦃\, rcvq := udp\_sock.rcvq \ @\ [(\textsc{Dgram\_error}(⦃\, e := \mathrm{EMSGSIZE}⦄))]⦄)⦄ \wedge oq' = oq$
else
$sock' = sock \,⦃\, es := \uparrow \mathrm{EMSGSIZE}⦄ \wedge oq' = oq)$

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_MSGSIZE.

---

deliver_in_icmp_3 **all: network nonurgent** **Receive** *ICMP_UNREACH_PORT* **etc for known socket**

$h_0 \xrightarrow{\tau} h \,⦃ socks := socks \oplus$
$[(sid, sock')];$
$iq := iq';$
$oq := oq'⦄$

$h_0 = h \,⦃\, socks := socks \oplus$
$[(sid, sock)];$
$iq := iq;$
$oq := oq⦄ \wedge$
$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{ICMP}\ icmp)) \wedge$
$icmp.t \in \{\mathrm{ICMP\_UNREACH}\ c \mid$
$c \in \{\mathrm{PROTOCOL}; \mathrm{PORT}; \mathrm{NET\_PROHIB}; \mathrm{HOST\_PROHIB}; \mathrm{FILTER\_PROHIB}\}\} \wedge$

$icmp.is_3 = \uparrow i_3 \wedge$
$i_3 \notin$ 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) $\rightarrow$
($\exists icmp_{seq}.icmp.seq = \uparrow icmp_{seq} \wedge$
if $tcp\_sock.cb.snd\_una \leq icmp_{seq} \wedge icmp_{seq} < tcp\_sock.cb.snd\_max$ **then**
if $tcp\_sock.st =$ SYN_SENT **then**
tcp_drop_and_close $h.arch(\uparrow$ ECONNREFUSED$)sock(sock', [\,])$ (* know from definition of tcp_drop_and_close that no segs will be emitted *)
else
$sock' = sock \wedge oq' = oq$
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
$sock' = sock \wedge oq' = oq) \parallel$
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 :≙ ↑(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**

$h_0 \xrightarrow{\tau} h$ ⦃$socks := socks \oplus$
$[(sid, sock')]$;
$iq := iq'$;
$oq := oq'$⦄

$h_0 = h$ ⦃ $socks := socks \oplus$
$[(sid, sock)]$;
$iq := iq$;
$oq := oq$⦄ $\wedge$
dequeue_iq(iq, iq', ↑(ICMP $icmp$)) $\wedge$
$icmp.t \in$ {ICMP_PARAMPROB c |
$c \in$ {BADHDR; NEEDOPT}} $\wedge$
$icmp.is_3 = \uparrow i_3 \wedge$
$i_3 \notin$ 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) $\rightarrow$
($\exists icmp_{seq}.icmp.seq = \uparrow icmp_{seq} \wedge$
if $tcp\_sock.cb.snd\_una \leq icmp_{seq} \wedge icmp_{seq} < tcp\_sock.cb.snd\_max$ **then**
if $tcp\_sock.st \in$ {CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED} $\wedge$
$tcp\_sock.cb.tt\_rexmt \neq * \wedge$ shift_of $tcp\_sock.cb.tt\_rexmt > 3 \wedge$
$tcp\_sock.cb.t\_softerror \neq *$ **then**
tcp_drop_and_close $h.arch(\uparrow$ ENOPROTOOPT$)sock(sock', outsegs) \wedge$
enqueue_and_ignore_fail $h.arch\ h.rttab\ h.ifds\ outsegs\ oq\ oq'$
else

$sock' = sock$ ⦇ $pr :=$ TCP_PROTO(tcp_sock
⦇ $cb := tcp\_sock.cb$ ⦇ $t\_softerror := \uparrow$ ENOPROTOOPT⦈⦈)⦈ $\wedge$
$oq' = oq$
else

$sock' = sock \wedge oq' = oq$) ‖
UDP_PROTO(udp_sock) $\rightarrow$
(**if** windows_arch $h.arch$ **then**
$sock' = sock$ ⦇ $pr :=$ UDP_PROTO(udp_sock
⦇ $rcvq := udp\_sock.rcvq$ @ [(DGRAM_ERROR(⦇ $e :=$ ENOPROTOOPT⦈))]⦈)⦈ $\wedge$
$oq' = oq$
else
$sock' = sock$ ⦇ $es := \uparrow$(ENOPROTOOPT)⦈ $\wedge$ $oq' = oq$))

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_PARAMPROB.

deliver_in_icmp_5 **all: network nonurgent** **Receive ICMP_SOURCE_QUENCH for known socket**

$h_0 \xrightarrow{\tau} h$ ⦇$socks := socks \oplus$
$[(sid, sock')]$;
$iq := iq'$⦈

$h_0 = h$ ⦇ $socks := socks \oplus$
$[(sid, sock)]$;
$iq := iq$⦈ $\wedge$
dequeue_iq($iq, iq', \uparrow$(ICMP $icmp$)) $\wedge$
$icmp.t =$ ICMP_SOURCE_QUENCH QUENCH $\wedge$
$icmp.is_3 = \uparrow i_3 \wedge$
$i_3 \notin$ 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) $\rightarrow$
($\exists icmp_{seq}.icmp.seq = \uparrow icmp_{seq} \wedge$
if $tcp\_sock.cb.snd\_una \leq icmp_{seq} \wedge icmp_{seq} < tcp\_sock.cb.snd\_max$ **then**
$sock' = sock$ ⦇ $pr :=$ TCP_PROTO(tcp_sock
⦇ $cb := tcp\_sock.cb$
⦇ $snd\_cwnd := 1 * tcp\_sock.cb.t\_maxseg$⦈⦈)⦈
(* Note the state of the TCP socket should be checked here. *)
(* Note it might be necessary to make an allowance for local/remote connection? *)
else
(* Note the case where it is a syncache entry is not dealt with here: a `syncache_unreach()` should be done instead *)
$sock' = sock$) ‖
UDP_PROTO(udp_sock) $\rightarrow$
(**if** windows_arch $h.arch$ **then**
$sock' = sock$ ⦇ $pr :=$ UDP_PROTO(udp_sock
⦇ $rcvq := udp\_sock.rcvq$ @ [(DGRAM_ERROR(⦇ $e :=$ EHOSTUNREACH⦈))]⦈)⦈
else
$sock' = sock$ ⦇ $es := \uparrow$(EHOSTUNREACH)⦈))

Description Corresponds to FreeBSD 4.6-RELEASE's PRC_QUENCH.

deliver_in_icmp_6 **all: network nonurgent** **Receive and ignore other ICMP**

$h \; \langle\!|iq := iq|\!\rangle \;\; \xrightarrow{\tau} \;\; h \; \langle\!|iq := iq'|\!\rangle$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{ICMP}\ icmp)) \wedge$
$(icmp.t \ \in \{\mathrm{ICMP\_TIME\_EXCEEDED\ INTRANS}; \mathrm{ICMP\_TIME\_EXCEEDED\ REASS}\} \vee$
$icmp.t \ \in \{\mathrm{ICMP\_UNREACH(OTHER}\ x) \mid x \ \in \ \mathit{UNIV}\} \vee$
$icmp.t \ \in \{\mathrm{ICMP\_SOURCE\_QUENCH(OTHER}\ x) \mid x \ \in \ \mathit{UNIV}\} \vee$
$icmp.t \ \in \{\mathrm{ICMP\_TIME\_EXCEEDED(OTHER}\ x) \mid x \ \in \ \mathit{UNIV}\} \vee$
$icmp.t \ \in \{\mathrm{ICMP\_PARAMPROB(OTHER}\ x) \mid x \ \in \ \mathit{UNIV}\})$

Description If ICMP_TIME_EXCEEDED (either INTRANS or REASS), or if a bad code is received, then ignore silently.

deliver_in_icmp_7 **all: network nonurgent** **Receive and ignore invalid or unmatched ICMP**

$h \; \langle\!|iq := iq|\!\rangle \;\; \xrightarrow{\tau} \;\; h \; \langle\!|iq := iq'|\!\rangle$

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathrm{ICMP}\ icmp)) \wedge$
$(icmp.t \ \in \{\mathrm{ICMP\_UNREACH}\ c \mid \neg\exists x.c = \mathrm{OTHER}\ x\} \vee$
$icmp.t \ \in \{\mathrm{ICMP\_PARAMPROB}\ c \mid c \ \in \{\mathrm{BADHDR}; \mathrm{NEEDOPT}\}\} \vee$
$icmp.t = \mathrm{ICMP\_SOURCE\_QUENCH\ QUENCH}) \wedge$
$(\textbf{if}\ \exists icmpmtu.icmp.t = \mathrm{ICMP\_UNREACH(NEEDFRAG}\ icmpmtu)\ \textbf{then}$
$\quad \exists i_3.icmp.is_3 = \uparrow i_3 \wedge i_3 \ \in \ \mathrm{IN\_MULTICAST}$
$\textbf{else}$
$\quad (icmp.is_3 = * \vee$
$\quad\quad \textbf{the}\ \ icmp.is_3 \ \in \ \mathrm{IN\_MULTICAST} \vee$
$\quad\quad \neg(\exists(\mathsf{sid}, s) :: (h.socks).$
$\quad\quad\quad\quad s.is_1 = icmp.is_3 \wedge s.is_2 = icmp.is_4 \wedge$
$\quad\quad\quad\quad s.ps_1 = icmp.ps_3 \wedge s.ps_2 = icmp.ps_4 \wedge$
$\quad\quad\quad\quad \mathrm{proto\_of}\ \ s.pr = icmp.proto)))$

Description If the ICMP is a type we handle, but the source IP is IP 0 0 00 or a multicast address, or there's no matching socket, then drop silently. ICMP_UNREACH NEEDFRAG is handled specially, since we do not care if it's IP 0 0 0 0, only if it's multicast.

Chapter 21

Host LTS: Network Input and Output

21.1 Input and Output (Network only)

21.1.1 Summary

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

21.1.2 Rules

deliver_in_99 **all: network nonurgent** **Really receive things**

$h\ \langle\!|iq := iq|\!\rangle \xrightarrow{msg} h\ \langle\!|iq := iq'|\!\rangle$

sane_msg $msg \wedge$
$\uparrow i_1 = msg.is_2 \wedge$
$i_1 \in$ local_ips$(h.ifds) \wedge$
enqueue_iq$(iq, msg, iq', queued)$

Description Actually receive a message from the wire into the input queue. Note that if it cannot be queued (because the queue is full), it is silently dropped.

We only accept messages that are for this host. We also assert that any message we receive is well-formed (this excludes elements of type *msg* that have no physical realisation).

Note the delay in in-queuing the datagram is not modelled here.

deliver_in_99a **all: network nonurgent** **Ignore things not for us**

$h\ \langle\!|iq := iq|\!\rangle \xrightarrow{msg} h\ \langle\!|iq := iq'|\!\rangle$

$\uparrow i_1 = msg.is_2 \wedge$
$i_1 \notin$ 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**

$h \ \langle\!| oq := oq |\!\rangle \xrightarrow{\overline{msg}} h \ \langle\!| oq := oq' |\!\rangle$

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**

$h \ \langle\!| iq := iq;$ $oq := oq |\!\rangle \xrightarrow{lbl} h \ \langle\!| iq := iq';$ $oq := oq' |\!\rangle$

dequeue_oq$(oq, oq', \uparrow msg) \wedge$
$(\exists i_2.msg.is_2 = \uparrow i_2 \wedge i_2 \in$ local_ips $h.ifds) \wedge$
$(lbl =$ **if** windows_arch $h.arch$ **then** τ
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 22

Host LTS: BSD Trace Records and Interface State Changes

22.1 Trace Records and Interface State Changes (BSD only)

22.1.1 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

22.1.2 Rules

trace_1 **all: misc nonurgent** **Trace TCPCB state, ESTABLISHED or later**

$$h \xrightarrow{\text{LH\_TRACE } tr} h$$

$sid \in \mathbf{dom}(h.socks) \wedge$
$tr = (flav, sid, quad, st, cb) \wedge$
$st \in \{\text{ESTABLISHED}; \text{FIN\_WAIT\_1}; \text{FIN\_WAIT\_2}; \text{CLOSING};$
$\quad \text{CLOSE\_WAIT}; \text{LAST\_ACK}; \text{TIME\_WAIT}\} \wedge$
$\text{tracesock\_eq } tr\ sid(h.socks[sid])$

Description This rule exposes certain of the fields of the socket and TCPCB, to allow open-box testing.

Note that although the label carries an entire TCPCB, only certain selected fields are constrained to be equal to the actual TCPCB. See tracesock_eq (p63) and tracecb_eq (p62) for details.

Checking trace equality is problematic as BSD generates trace records that fall logically inbetween the atomic transitions in this model. This happens frequently when in a state before ESTABLISHED. We only check for equality when we are in ESTABLISHED or later states.

trace_2 **all: misc nonurgent** **Trace TCPCB state, pre-ESTABLISHED**

$$h \xrightarrow{\text{LH\_TRACE } tr} h$$

$sid \in \mathbf{dom}(h.socks) \wedge$
$tr = (flav, sid, quad, st, cb) \wedge$
$st \notin \{\text{ESTABLISHED}; \text{FIN\_WAIT\_1}; \text{FIN\_WAIT\_2}; \text{CLOSING};$
$\quad \text{CLOSE\_WAIT}; \text{LAST\_ACK}; \text{TIME\_WAIT}\} \wedge$

$(st = \mathrm{CLOSED} \lor$ (* BSD emits one of these each time a tcpcb is created, eg at end of 3WHS *)
$((\exists sock\ tcp\_sock.$
$\quad sock = (h.socks[sid]) \land$
$\quad \mathrm{proto\_of}\ \ sock.pr = \mathrm{PROTO\_TCP} \land$
$\quad tcp\_sock = \mathrm{tcp\_sock\_of}\ \ sock \land$
$\quad (\mathbf{case}\ \ quad\ \ \mathbf{of}$
$\qquad \uparrow(is_1, ps_1, is_2, ps_2) \rightarrow \mathbf{if}\ \ flav = \mathrm{TA\_DROP} \lor tcp\_sock.st = \mathrm{CLOSED}\ \ \mathbf{then}\ \ \mathbf{T}$
$\qquad\qquad \mathbf{else}$
$\qquad\qquad\quad is_1 = sock.is_1 \land ps_1 = sock.ps_1 \land is_2 = sock.is_2 \land ps_2 = sock.ps_2 \parallel$
$\qquad * \rightarrow \mathbf{T}) \land$
$\quad (st = tcp\_sock.st \lor tcp\_sock.st = \mathrm{CLOSED}))))$

---

interface_1 **all: misc nonurgent** **Change connectivity**

$$h\ \langle\!|ifds := ifds|\!\rangle \xrightarrow{\textsc{Lh\_interface}(ifid,\ up)} h\ \langle\!|ifds := ifds'|\!\rangle$$

$ifid\ \in \mathbf{dom}(ifds) \land$
$ifds' = ifds \oplus (ifid, (ifds[ifid])\langle\!|\ up := up|\!\rangle)$

Description Allow interfaces to be externally brought up or taken down.

Chapter 23

Host LTS: Time Passage

23.1 Time Passage auxiliaries (TCP and UDP)

Time passage is a *function*, completely deterministic. Any nondeterminism must occur as a result of a tau (or other) transition.

In the present semantics, time passage merely:

1. decrements all timers uniformly
2. prevents time passage if a timer reaches zero
3. prevents time passage if an urgent action is enabled.

We model the first two points with functions *Time_Pass_*∗, for various types ∗. These functions return an option type: if the result is NONE then time may not pass for the given duration. Essentially they pick out everything in a host state of type $'a$ timed, and do something with it.

We treat the last point in the rule *epsilon_1* (p348) itself, below.

23.1.1 Summary

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

23.1.2 Rules

– **time passes for an $'a$ timed option value :**
(Time_Pass_timedoption : duration → $'a$ timed option → $'a$ timed option option)
dur x0
= **case** *x0* **of**
  ∗ → ↑ ∗ ‖
  ↑ *x* → (**case** Time_Pass_timed *dur x* **of**
      ∗ → ∗ ‖
      ↑ *x0*′ → ↑(↑ *x0*′))

– **time passes for a tcp control block :**

(Time_Pass_tcpcb : duration $\rightarrow$ tcpcb $\rightarrow$ tcpcb set option)(* recall: 'a set == 'a -> bool *)
$dur\ cb$
$= \mathbf{let}\ \ tt\_rexmt' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_rexmt$
$\mathbf{and}\ \ tt\_keep' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_keep$
$\mathbf{and}\ \ tt\_2msl' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_2msl$
$\mathbf{and}\ \ tt\_delack' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_delack$
$\mathbf{and}\ \ tt\_conn\_est' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_conn\_est$
$\mathbf{and}\ \ tt\_fin\_wait\_2' = \text{Time\_Pass\_timedoption}\ \ dur\ cb.tt\_fin\_wait\_2$
$\mathbf{and}\ \ ts\_recent's = \text{Time\_Pass\_timewindow}\ \ dur\ cb.ts\_recent$
$\mathbf{and}\ \ t\_badrxtwin's = \text{Time\_Pass\_timewindow}\ \ dur\ cb.t\_badrxtwin$
$\mathbf{and}\ \ t\_idletime's = \text{Time\_Pass\_stopwatch}\ \ dur\ cb.t\_idletime$
$\mathbf{in}$
$\mathbf{if}\ \ \mathbf{is\_some}\ \ tt\_rexmt' \wedge$
$\mathbf{is\_some}\ \ tt\_keep' \wedge$
$\mathbf{is\_some}\ \ tt\_2msl' \wedge$
$\mathbf{is\_some}\ \ tt\_delack' \wedge$
$\mathbf{is\_some}\ \ tt\_conn\_est' \wedge$
$\mathbf{is\_some}\ \ tt\_fin\_wait\_2'$
$\mathbf{then}$
$\uparrow(\lambda cb'.$
$\mathbf{choose}\ \ ts\_recent' :: ts\_recent's.$
$\mathbf{choose}\ \ t\_badrxtwin' :: t\_badrxtwin's.$
$\mathbf{choose}\ \ t\_idletime' :: t\_idletime's.$
$cb' =$
$cb\ \langle\!|$ (* not going to list everything here; too much! *)
$tt\_rexmt := \mathbf{the}\ \ tt\_rexmt';$
$tt\_keep := \mathbf{the}\ \ tt\_keep';$
$tt\_2msl := \mathbf{the}\ \ tt\_2msl';$
$tt\_delack := \mathbf{the}\ \ tt\_delack';$
$tt\_conn\_est := \mathbf{the}\ \ tt\_conn\_est';$
$tt\_fin\_wait\_2 := \mathbf{the}\ \ tt\_fin\_wait\_2';$
$ts\_recent := ts\_recent';$
$t\_badrxtwin := t\_badrxtwin';$
$t\_idletime := t\_idletime'$
$|\!\rangle)$
$\mathbf{else}$
$*$

$-$ **time passes for a socket :**
(Time_Pass_socket : duration $\rightarrow$ socket $\rightarrow$ socket set option)
$dur\ s$
$= \mathbf{case}\ \ s.pr\ \ \mathbf{of}\ \ \text{UDP\_PROTO}(udp) \rightarrow \uparrow\{s\}$
$\|\ \text{TCP\_PROTO}(tcp\_s) \rightarrow$
$\mathbf{let}\ \ cb's = \text{Time\_Pass\_tcpcb}\ \ dur\ tcp\_s.cb$
$\mathbf{in}$
$\mathbf{if}\ \ \mathbf{is\_some}\ \ cb's$
$\mathbf{then}$
$\uparrow(\lambda s'.$
$\mathbf{choose}\ \ cb' :: \mathbf{the}\ \ cb's.$
$s' =$
$s\ \langle\!|$ (* fid unchanged *)
(* sf unchanged *)
(* is1,ps1,is2,ps2 unchanged *)
(* es unchanged *)
$pr := \text{TCP\_PROTO}(tcp\_s\ \langle\!|\ cb := cb' |\!\rangle)$
$|\!\rangle)$

else
$*$

– **apply** f **to range of finite map, and succeed if each application succeeds :**
(fmap_every : $('a \to 'b$ option$) \to ('c \mapsto 'a) \to ('c \mapsto 'b)$ option$)$
$f\ fm =$
let $fm' = f\ o\_f\ fm$
in
if $* \in \mathbf{rng}(fm')$
then $*$
else $\uparrow(\mathbf{the}\ o\_f\ fm')$

– **apply** f **to range of finite map, and succeed if each application succeeds :**
(fmap_every_pred : $('a \to 'b$ set option$) \to ('c \mapsto 'a) \to ('c \mapsto 'b)$set option$)$
$f\ fm =$
if $\exists y. y \in \mathbf{rng}(fm) \wedge f\ y = *$ **then**
$*$
else
$\uparrow\{fm' \mid \mathbf{dom}(fm) = \mathbf{dom}(fm') \wedge$
$\forall x. x \in \mathbf{dom}(fm) \implies fm'[x] \in (\mathbf{the}(f(fm[x])))\}$

– **time passes for a host :**
(Time_Pass_host : duration $\to$ host $\to$ host set option)
$dur\ h$
$=$ **let** $ts' =$ fmap_every(Time_Pass_timed dur)$h.ts$
and $socks's =$ fmap_every_pred(Time_Pass_socket dur)$h.socks$
and $iq' =$ Time_Pass_timed $dur\ h.iq$
and $oq' =$ Time_Pass_timed $dur\ h.oq$
and $ticks's =$ Time_Pass_ticker $dur\ h.ticks$
in
if is_some $ts' \wedge$
is_some $socks's \wedge$
is_some $iq' \wedge$
is_some oq'
then
$\uparrow(\lambda h'.$
choose $socks' ::$ **the** $socks's$.
choose $ticks' :: ticks's$.
$h' =$
h ⟨[(* arch unchanged *)
(* ifds unchanged *)
$ts :=$ **the** ts';
(* files unchanged *)
$socks := socks'$;
(* listen unchanged *)
(* bound unchanged *)
$iq :=$ **the** iq';
$oq :=$ **the** oq';
$ticks := ticks'$
(* fds unchanged *)

$\rrbracket)$
else
$*$

23.2 Host transitions with time (TCP and UDP)

We now build the relation $\Rightarrow$, which includes time transitions, from the relation $\rightarrow$, which is instantaneous. This avoids circularity (or at best inductiveness) in the definition of the transition relation.

23.2.1 Summary

epsilon_1	**all: misc nonurgent**	Time passes
epsilon_2	**all: misc nonurgent**	Inductively defined time passage
rn	**rp: rc**	

23.2.2 Rules

epsilon_1 **all: misc nonurgent** **Time passes**

$$h \quad \xRightarrow{dur} \quad h'$$

$$\begin{array}{l}
\textbf{let } hs' = \mathrm{Time_Pass_host}\ dur\ h\ \textbf{in}\\
\quad \textbf{is_some}\ hs' \wedge\\
\quad h' \in (\textbf{the}\ hs') \wedge\\
\quad \neg(\exists rn\ rp\ rc\ lbl\ h'.rn/*rp, rc*/h \xrightarrow{lbl} h' \wedge \mathrm{is_urgent}\ rc)
\end{array}$$

Description Allow time to pass for *dur* seconds. This is only enabled if the host state is not urgent, i.e. if no urgent rule can fire. Notice that, apart from when a timer becomes zero, a host state never becomes urgent due merely to time passage. This means we need only test for urgency at the beginning of the time interval, not throughout it.

epsilon_2 **all: misc nonurgent** **Inductively defined time passage**

$$h \quad \xRightarrow{dur} \quad h'$$

$$\begin{array}{l}
(\exists h_1\ h_2\ dur'\ dur''.\\
\quad dur' < dur \wedge\\
\quad (\exists rn\ rp\ rc.rn/*rp, rc*/h \xRightarrow{dur'} h_1) \wedge\\
\quad (\exists rn\ rp\ rc.rn/*rp, rc*/h_1 \xRightarrow{\tau} h_2) \wedge\\
\quad dur' + dur'' = dur \wedge\\
\quad (\exists rn\ rp\ rc.rn/*rp, rc*/h_2 \xRightarrow{dur''} h')\\
)
\end{array}$$

Description Combine time passage and τ transitions.

rn **rp: rc**

$h \quad \overset{lbl}{\Longrightarrow} \quad h'$

$rn / * \, rp, rc * /$
h
$\xrightarrow{lbl}$
h'

Description Embed all non-time transitions in the full LTS

Part XIV

TCP1_evalSupport

Chapter 24

Initial state

This file defines a function to construct certain initial host states for use in automated trace checking, along with other constants used in typical traces. The interfaces, routing table and some host fields are taken from the `initial_host` line at the start of a valid trace.

24.1 Initial state (TCP and UDP)

The initial state of a host.

24.1.1 Summary

simple_ifd_eth	simple ethernet interface
simple_ifd_lo	simple loopback interface
simple_rttab	simple routing table
tid_initial	initial thread id
simple_host	simple host state
dummy_cb	
dummy_socket	minimal socket
dummy_sockets	
initial_host	function to construct an initial host for trace checking

24.1.2 Rules

– **simple ethernet interface :**

$\mathrm{simple\_ifd\_eth}\ \ i = (\mathrm{ETH}\ 0, \langle\!|\ \mathit{ipset} := \{i\}; \mathit{primary} := i; \mathit{netmask} := \mathrm{NETMASK}\ 24; \mathit{up} := \mathbf{T} |\!\rangle)$

– **simple loopback interface :**

$\mathrm{simple\_ifd\_lo} = (\mathrm{LO}, \langle\!|\ \mathit{ipset} := \mathrm{LOOPBACK\_ADDRS}; \mathit{primary} := \mathrm{ip\_localhost};$
$\quad \mathit{netmask} := \mathrm{NETMASK}\ 8; \mathit{up} := \mathbf{T} |\!\rangle)$

– **simple routing table :**

$\mathrm{simple\_rttab} = [\langle\!|\ \mathit{destination\_ip} := \mathrm{ip\_localhost};$
$\quad \mathit{destination\_netmask} := \mathrm{NETMASK}\ 8;$
$\quad \mathit{ifid} := \mathrm{LO} |\!\rangle;$
$\quad \langle\!|\ \mathit{destination\_ip} := \mathrm{IP}\ \ 0\ 0\ 0\ 0;$

$\quad destination\_netmask := \mathrm{NETMASK}\ 0;$
$\quad ifid := \mathrm{ETH}\ 0 \rrbracket]$

– initial thread id :
tid_initial = TID 0

– simple host state :

$$
\begin{aligned}
&\text{simple\_host}\ \ i\ tick0\ remdr0 = \\
&\quad \llparenthesis\ arch := \mathrm{F{\scriptsize REE}BSD\_4\_6\_RELEASE}; \\
&\quad\ privs := \mathbf{F}; \\
&\quad\ ifds := \emptyset \oplus [\text{simple\_ifd\_lo}; \text{simple\_ifd\_eth}\ \ i]; \\
&\quad\ rttab := \text{simple\_rttab}; \\
&\quad\ ts := \emptyset \oplus (\text{tid\_initial} \mapsto (\mathrm{R{\scriptsize UN}})_{\text{never\_timer}}); \\
&\quad\ files := \emptyset; \\
&\quad\ socks := \emptyset; \\
&\quad\ \mathsf{listen} := [\,]; \\
&\quad\ bound := [\,]; \\
&\quad\ iq := ([\,])_{\text{never\_timer}}\,; \\
&\quad\ oq := ([\,])_{\text{never\_timer}}\,; \\
&\quad\ bndlm := \text{bandlim\_state\_init}; \\
&\quad\ ticks := \mathrm{T{\scriptsize ICKER}}(tick0, remdr0, \text{tickintvlmin}, \text{tickintvlmax}); \\
&\quad\ fds := \emptyset \rrparenthesis
\end{aligned}
$$

– :

$$
\begin{aligned}
\text{dummy\_cb} = \llparenthesis\ & tt\_rexmt := *; \\
& tt\_2msl := *; \\
& tt\_conn\_est := *; \\
& tt\_delack := *; \\
& tt\_keep := *; \\
& tt\_fin\_wait\_2 := *; \\
& t\_idletime := \mathrm{S{\scriptsize TOPWATCH}}(0, 1, 1); \\
& t\_badrxtwin := \mathrm{T{\scriptsize IME}W{\scriptsize INDOW}C{\scriptsize LOSED}}; \\
& ts\_recent := \mathrm{T{\scriptsize IME}W{\scriptsize INDOW}C{\scriptsize LOSED}} \rrparenthesis
\end{aligned}
$$

– minimal socket :

$$
\begin{aligned}
&\text{dummy\_socket}(is, p) = \\
&\qquad \llparenthesis\ fid := *; \\
&\qquad\quad sf := \llparenthesis\ b := \lambda x.\mathbf{F}; n := \lambda x.0; t := \lambda x.\infty \rrparenthesis; \\
&\qquad\quad is_1 := is; \\
&\qquad\quad ps_1 := \uparrow p; \\
&\qquad\quad is_2 := *; \\
&\qquad\quad ps_2 := *; \\
&\qquad\quad pr := \mathrm{TCP\_PROTO}(\llparenthesis\ st := \mathrm{LISTEN}; \\
&\qquad\qquad\qquad cb := \text{dummy\_cb}; \\
&\qquad\qquad\qquad lis := \uparrow \llparenthesis\ q_0 := [\,]; q := [\,]; qlimit := 10 \rrparenthesis \\
&\qquad\qquad\quad \rrparenthesis) \\
&\qquad \rrparenthesis
\end{aligned}
$$

Description This is a pretty minimally-defined socket, just enough to say "this port is bound".

– :

$\text{dummy\_sockets}\ n[\,] = [\,] \wedge$
$\text{dummy\_sockets}\ n(p :: ps) = (\text{SID}\ n, \text{dummy\_socket}\ \ p) :: \text{dummy\_sockets}(n+1)ps$

– function to construct an initial host for trace checking :

$\text{initial\_host}(i : \mathsf{ip})(t : \mathsf{tid})(arch : arch)(ispriv : \mathsf{bool})$
$(heldports : (\mathsf{ip\ \ option\#port})\mathsf{list})(ifaces : (ifid\#\mathsf{ifd})\mathsf{list})$
$(rt : routing\_table)$
$(init\_tick : \text{ts\_seq})$
$(init\_tick\_remdr : \text{duration})$
$= \text{simple\_host}\ \ i\ init\_tick\ init\_tick\_remdr\ \langle\!|$
$arch := arch;$
$privs := ispriv;$
$ifds := \emptyset \oplus ifaces;$
$rttab := rt;$
$ts := \emptyset \oplus (t \mapsto (\text{RUN})_{\text{never\_timer}});$
$fds := \textbf{case}\ \ arch\ \ \textbf{of}$
(* per architecture, note down FDs preallocated for internal use by OCaml or the test harness *)
$\text{LINUX\_2\_4\_20\_8} \rightarrow$
$\emptyset \oplus [(\text{FD}\ 0, \text{FID}\ 0);$
$(\text{FD}\ 1, \text{FID}\ 0);$
$(\text{FD}\ 2, \text{FID}\ 0);$
$(\text{FD}\ 3, \text{FID}\ 0);$
$(\text{FD}\ 4, \text{FID}\ 0);$
$(\text{FD}\ 5, \text{FID}\ 0);$
$(\text{FD}\ 6, \text{FID}\ 0);$
$(\text{FD}\ 1000, \text{FID}\ 0)$
$]$
$\parallel \text{FREEBSD\_4\_6\_RELEASE} \rightarrow$
$\emptyset \oplus [(\text{FD}\ 0, \text{FID}\ 0);$
$(\text{FD}\ 1, \text{FID}\ 0);$
$(\text{FD}\ 2, \text{FID}\ 0);$
$(\text{FD}\ 3, \text{FID}\ 0);$
$(\text{FD}\ 4, \text{FID}\ 0);$
$(\text{FD}\ 5, \text{FID}\ 0);$
$(\text{FD}\ 6, \text{FID}\ 0);$
$(\text{FD}\ 7, \text{FID}\ 0)$
$]$
$\parallel \text{WINXP\_PROF\_SP1} \rightarrow$
$\emptyset$; (* Windows FDs are not allocated in order, so there's no need to specify anything here. *)
$files := \emptyset \oplus (\text{FID}\ 0,$
$\text{FILE}(\text{FT\_CONSOLE}, \langle\!|\ b := \lambda x.\mathbf{F} |\!\rangle));$
$socks := \emptyset \oplus (\text{dummy\_sockets}\ \ 0\ heldports)$
$|\!\rangle$

Index