1 Host transition rules

socket_1 **succeed**

$$
\begin{aligned}
& h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \\
\xrightarrow{tid \cdot (\text{socket}())} \; & h \text{ with } \langle\!| ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK} fd))_{\text{dsch}}); \\
& \quad s := (\textsc{Sock}(fd, *, *, *, *, *, \textsc{Flags}(\mathbf{F}, \mathbf{F}), [\,]) :: h.s) \\
& \quad |\!\rangle
\end{aligned}
$$

$fd \notin \text{sockfds}\, h.s$

In BSD, this would be [[Flags(T,F)]].

socket_2 **badfail** **as no file descriptors or file table space**

$$
\begin{aligned}
& h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \\
\xrightarrow{tid \cdot \text{socket}()} \; & h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}})
\end{aligned}
$$

$e \in \{\text{EMFILE}; \text{ENFILE}\}$

The errors are \sEMFILE{} and \sENFILE.

bind_1 **succeed** **autobinding**

$$
\begin{aligned}
& FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \\
\xrightarrow{tid \cdot \text{bind}(fd, *, *)} \; & FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, \textsc{Sock}(fd, *, \uparrow p_1', *, *, es, f, mq))
\end{aligned}
$$

F_context FC $\wedge$
$p_1' \in$ unused $FC \cap$ ephemeral

bind_2 **succeed** **autobinding**

$$
\begin{aligned}
& FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \\
\xrightarrow{tid \cdot \text{bind}(fd, \uparrow i, *)} \; & FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, \textsc{Sock}(fd, \uparrow i, \uparrow p_1', *, *, es, f, mq))
\end{aligned}
$$

F_context FC $\wedge$
$i \in ifds$ $\wedge$
$p_1' \in$ unused $FC \cap$ ephemeral

bind_3 **succeed** **autobinding**

$$
\begin{aligned}
& FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \\
\xrightarrow{tid \cdot \text{bind}(fd, *, \uparrow p)} \; & FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, \textsc{Sock}(fd, *, \uparrow p, *, *, es, f, mq))
\end{aligned}
$$

F_context FC $\wedge$
$p \notin$ privileged $\wedge$
$\forall s. s \in FC \wedge (s.ps_1 = \uparrow p) \implies$
$\quad f.reuseaddr \wedge s.f.reuseaddr$

Note that the [[i1]] doesn't get chosen until a datagram is sent, as in \cite[p.92]{stevens1998a}.

bind_4 **succeed**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Run})_d, \textsc{Sock}(\mathit{fd}, *, *, *, *, \mathit{es}, f, \mathit{mq})) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{bind}(\mathit{fd},\uparrow i,\uparrow p)} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Ret}(\mathrm{OK}()))_{\mathrm{dsch}}, \textsc{Sock}(\mathit{fd}, \uparrow i, \uparrow p, *, *, \mathit{es}, f, \mathit{mq})) \end{array}$$

$$\begin{array}{l} \text{F\_context } FC \wedge \\ i \in \mathit{ifds} \wedge \\ p \notin \text{privileged} \wedge \\ \forall s. s \in FC \implies \\ \quad (s.ps_1 = \uparrow p) \wedge ((s.is_1 = *) \vee (s.is_1 = \uparrow i)) \implies \\ \quad f.\mathit{reuseaddr} \wedge s.f.\mathit{reuseaddr} \end{array}$$

bind_5 **fail** **as we don't have access for port p**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Run})_d, s \text{ with } \mathit{fd} := \mathit{fd}) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{bind}(\mathit{fd},\mathit{is},\uparrow p)} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Ret}(\text{FAIL EACCES}))_{\mathrm{dsch}}, s \text{ with } \mathit{fd} := \mathit{fd}) \end{array}$$

$$\begin{array}{l} \text{F\_context } FC \wedge \\ p \in \text{privileged} \end{array}$$

On Linux, [[is]] is written into the [[Sock(...)]]; we choose \emph{not} to do so because it would violate our tidy [[socket]] invariant. Since the socket is not placed on the delivery list, we can hide this by letting our wrapper for [[getsockname]] test the local port, and, if it is [[*]], return [[*]] for local [[is]].

bind_6 **fail** **as port p is already in use**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Run})_d, s \text{ with } \mathit{fd} := \mathit{fd}) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{bind}(\mathit{fd},\mathit{is},\uparrow p)} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Ret}(\text{FAIL EADDRINUSE}))_{\mathrm{dsch}}, s \text{ with } \mathit{fd} := \mathit{fd}) \end{array}$$

$$\begin{array}{l} \text{F\_context } FC \wedge \\ \neg(\forall s'. s' \in FC \implies \\ \quad (s'.ps_1 = \uparrow p) \wedge \\ \quad ((s'.is_1 = *) \vee (is = *) \vee (is = s'.is_1)) \implies \\ \quad s.f.\mathit{reuseaddr} \wedge s'.f.\mathit{reuseaddr}) \end{array}$$

bind_7 **fail** **as i is not one of our ip addresses**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Run})_d, s \text{ with } \mathit{fd} := \mathit{fd}) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{bind}(\mathit{fd},\uparrow i,\mathit{ps})} FC(\mathit{ifds}, \mathit{tid}, (\textsc{Ret}(\text{FAIL EADDRNOTAVAIL}))_{\mathrm{dsch}}, s \text{ with } \mathit{fd} := \mathit{fd}) \end{array}$$

$$\begin{array}{l} \text{F\_context } FC \wedge \\ i \notin \mathit{ifds} \end{array}$$

bind_8 **fail** **as the socket has a non-* port already**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } \langle\!|fd := fd; ps_1 := \uparrow p_1|\!\rangle) \xrightarrow{tid \cdot \text{bind}(fd, is, ps)} FC(ifds, tid, (\textsc{Ret}(\text{FAIL EINVAL}))_{\text{dsch}}, s \text{ with } \langle\!|fd := fd; ps_1 := \uparrow p_1|\!\rangle)$$

F_context FC

bind_9 **fail** **as there are no ephemeral ports left**

$$FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \xrightarrow{tid \cdot \text{bind}(fd, is, *)} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, \textsc{Sock}(fd, *, *, *, *, es, f, mq))$$

F_context FC $\wedge$
(unused $FC \cap$ ephemeral $= \emptyset$) $\wedge$
($e \in$ ephemeralErrors)

This error is awkward to test, as the test machines run out of files ([[ENFILE]] from [[socket]]) at about 3500, before the ephemeral ports are exhausted. The error set [[ephemeralErrors = \{EAGAIN,EADDRINUSE\}]] is therefore speculative. \mignore{Looking at the kernel, the tcp port rover has an explicit check for running out, but I (can't see the udp port rover in udpv4getport in net/ipv4/udp.c doing anything sensible. The error set is therefore a guess. We may be able to test by using sysctl to cut down the number of ephemeral ports. (a test machine crashes when we try to do that).}

connect_1 **succeed** **autobinding if necessary**

$$FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, ps_1, *, *, es, f, mq)) \xrightarrow{tid \cdot \text{connect}(fd, i, ps)} FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, \textsc{Sock}(fd, \uparrow i_1, \uparrow p_1', \uparrow i, ps, es, f, mq))$$

F_context FC $\wedge$
$i_1 \in$ outroute$(ifds, i)$ $\wedge$
$p_1' \in$ autobind(ps_1, FC)

connect_2 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, f, mq)) \xrightarrow{tid \cdot \text{connect}(fd, i, ps)} FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, \textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, ps, es, f, mq))$$

F_context FC

connect_3 **fail** **as there are no ephemeral ports left**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RUN})_d, \mathrm{SOCK}(fd, *, *, *, *, es, f, mq)) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{connect}(fd,i,ps)} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{dsch}}, s \text{ with } fd := fd) \end{array}$$

F_context FC $\wedge$
(unused $FC \cap$ ephemeral $= \emptyset$) $\wedge$
$e \in$ ephemeralErrors $\wedge$
socket_ok $\mathit{ifds}(s$ with $fd := fd)$

See remark for [[bind.9]]. The rule leaves the socket in a indeterminate state, as we do not want to specify exactly when the \texttt{getport} is called.

disconnect_1 **succeed**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RUN})_d, \mathrm{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, es, f, mq)) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{disconnect}fd} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RET}(\mathrm{OK}()))_{\mathrm{dsch}}, \mathrm{SOCK}(fd, *, \uparrow p_1, *, *, es, f, mq)) \end{array}$$

F_context FC

Note that this leaves the local port in place, perhaps surprisingly.

disconnect_2 **succeed** **but also autobind**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RUN})_d, \mathrm{SOCK}(fd, *, *, *, *, es, f, mq)) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{disconnect}fd} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RET}(\mathrm{OK}()))_{\mathrm{dsch}}, \mathrm{SOCK}(fd, *, \uparrow p_1, *, *, es, f, mq)) \end{array}$$

F_context FC $\wedge$
$p_1 \in$ unused $FC \cap$ ephemeral

This autobind may be surprising, but the resulting state \emph{can} receive msgs on [[*,p1]].

disconnect_3 **fail** **as there are no ephemeral ports left**

$$\begin{array}{l} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RUN})_d, \mathrm{SOCK}(fd, *, *, *, *, es, f, mq)) \\ \xrightarrow{\mathit{tid}\cdot\mathrm{disconnect}fd} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{RET}(\mathrm{FAIL}\ e))_{\mathrm{dsch}}, s \text{ with } fd := fd) \end{array}$$

F_context FC $\wedge$
(unused $FC \cap$ ephemeral $= \emptyset$) $\wedge$
$e \in$ ephemeralErrors $\wedge$
socket_ok $\mathit{ifds}(s$ with $fd := fd)$

See remark for [[connect_3]].

getsockname_1 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } fd := fd)$$
$$\xrightarrow{tid \cdot \text{getsockname} fd} FC(ifds, tid, (\textsc{Ret}(\text{OK}(s.is_1, s.ps_1)))_{\text{dsch}}, s \text{ with } fd := fd)$$

F_context FC

See comment at [[bind.5]] ([[EACCES]]). This involves a thin wrapper over the C system call; if we get [[(SOME is,*)]] from the C system call then return [[(*,*)]]. This [[ps1=*]] case of the rule covers for the weird Linux behaviour of a failing bind due to [[EACCES]] or [[EADDRINUSE]].

getpeername_1 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } fd := fd)$$
$$\xrightarrow{tid \cdot \text{getpeername} fd} FC(ifds, tid, (\textsc{Ret}(\text{OK}(s.is_2, s.ps_2)))_{\text{dsch}}, s \text{ with } fd := fd)$$

F_context FC

This involves a thin wrapper over the C system call: if we get [[ENOTCONN]] then return [[(*,*)]].

geterr_1 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } fd := fd)$$
$$\xrightarrow{tid \cdot \text{geterr} fd} FC(ifds, tid, (\textsc{Ret}(\text{OK} s.es))_{\text{dsch}}, s \text{ with } \langle\!| fd := fd; es := * |\!\rangle)$$

F_context FC

geterrfd is our name for the C call `getsockopt(fd,SOL_SOCKET,SO_ERROR,...)`.

getsockopt_1 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } fd := fd)$$
$$\xrightarrow{tid \cdot \text{getsockopt}(fd, opt)} FC(ifds, tid, (\textsc{Ret}(\text{OK} b))_{\text{dsch}}, s \text{ with } fd := fd)$$

F_context FC $\wedge$
$((opt = \text{SO\_BSDCOMPAT}) \implies (b = s.f.bsdcompat)) \wedge$
$((opt = \text{SO\_REUSEADDR}) \implies (b = s.f.reuseaddr))$

setsockopt_1 **succeed**

$$FC(ifds, tid, (\textsc{Run})_d, s \text{ with } fd := fd)$$
$$\xrightarrow{tid \cdot \text{setsockopt}(fd, opt, b)} FC(ifds, tid, (\textsc{Ret}(\text{OK}()))_{\text{dsch}}, s \text{ with } \langle\!| fd := fd; f := f' |\!\rangle)$$

F_context FC $\wedge$
$((opt = \text{SO\_BSDCOMPAT}) \implies (f' = s.f \text{ with } bsdcompat := b)) \wedge$
$((opt = \text{SO\_REUSEADDR}) \implies (f' = s.f \text{ with } reuseaddr := b))$

sendto_1 **succeed** **autobinding**

$$
\begin{aligned}
& h \text{ with } (\!| ts := ts \oplus (tid \mapsto (\textsc{Run})_d); \\
& \qquad s := SC(s \text{ with } es := *) |\!) \\
\xrightarrow{tid\cdot\text{sendto}(s.fd, ips, data, nb)} & h \text{ with } (\!| ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK}()))_{\text{dsch}}); \\
& \qquad s := SC(s \text{ with } (\!| es := *; ps_1 := \uparrow p_1' |\!)); \\
& \qquad oq := oq'; oqf := oqf' |\!)
\end{aligned}
$$

$$
\begin{aligned}
& \text{socklist\_context } SC \wedge \\
& p_1' \in \text{autobind}(s.ps_1, SC) \wedge \\
& (oq', oqf', \mathbf{T}) \in \text{dosend}(h.ifds, (ips, data), \\
& \qquad (s.is_1, \uparrow p_1', s.is_2, s.ps_2), \\
& \qquad h.oq, h.oqf) \wedge \\
& \text{string\_size } data \leq \text{UDPpayloadMax} \wedge \\
& ((ips \neq *) \vee (s.is_2 \neq *))
\end{aligned}
$$

sendto_2 **enter2** **entering Sendto2 state**

$$
\begin{aligned}
& h \text{ with } (\!| ts := ts \oplus (tid \mapsto (\textsc{Run})_d); \\
& \qquad s := SC(s \text{ with } es := *) |\!) \\
\xrightarrow{tid\cdot\text{sendto}(s.fd, ips, data, \mathbf{F})} & h \text{ with } (\!| ts := \\
& \qquad \mathbf{fupdate}\ ts \\
& \qquad\quad (tid, \textsc{Timed}(\textsc{Sendto2}(s.fd, ips, s.is_1, \uparrow p_1', s.is_2, s.ps_2, \\
& \qquad\qquad data), \\
& \qquad\qquad \infty)); \\
& \qquad s := SC(s \text{ with } (\!| es := *; ps_1 := \uparrow p_1' |\!)); \\
& \qquad oq := oq'; \\
& \qquad oqf := oqf' |\!)
\end{aligned}
$$

$$
\begin{aligned}
& \text{socklist\_context } SC \wedge \\
& p_1' \in \text{autobind}(s.ps_1, SC) \wedge \\
& (oq', oqf', \mathbf{F}) \in \text{dosend}(h.ifds, (ips, data), \\
& \qquad (s.is_1, \uparrow p_1', s.is_2, s.ps_2), \\
& \qquad h.oq, h.oqf) \wedge \\
& ((ips \neq *) \vee (s.is_2 \neq *))
\end{aligned}
$$

Note how the blocking call to sendto records the values of the socket's four address fields so that messages may still be delivered, even if the socket is later disconnected.

sendto_3 **<u>fail</u> as would block**

$$\begin{array}{l} h \text{ with } \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d); \\ \qquad s := SC(s \text{ with } es := *) |\!\rangle \\ \xrightarrow{tid \cdot \text{sendto}(s.fd, ips, data, \mathbf{T})} h \text{ with } \langle\!| ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EAGAIN}))_{\text{dsch}}); \\ \qquad s := SC(s \text{ with } \langle\!| es := *; ps_1 := \uparrow p_1' |\!\rangle); \\ \qquad oq := oq'; \\ \qquad oqf := oqf' |\!\rangle \end{array}$$

socklist_context SC $\wedge$
$p_1' \in \text{autobind}(s.ps_1, SC)$ $\wedge$
$(oq', oqf', \mathbf{F}) \in \text{dosend}(h.ifds, (ips, data),$
$\qquad (s.is_1, \uparrow p_1', s.is_2, s.ps_2),$
$\qquad h.oq, h.oqf)$ $\wedge$
$((ips \neq *) \vee (s.is_2 \neq *))$

Note that in Linux [[\tscon{EWOULDBLOCK}]] and [[EAGAIN]] are aliased.

sendto_4 **<u>fail</u> as socket unconnected**

$$\begin{array}{l} FC(ifds, tid, (\text{RUN})_d, \text{SOCK}(fd, is_1, ps_1, *, *, *, f, mq)) \\ \xrightarrow{tid \cdot \text{sendto}(fd, *, data, nb)} FC(ifds, tid, (\text{RET}(\text{FAIL ENOTCONN}))_{\text{dsch}}, \text{SOCK}(fd, is_1, \uparrow p_1', *, *, *, f, mq)) \end{array}$$

F_context FC $\wedge$
$p_1' \in \text{autobind}(ps_1, FC)$

\cite{stevens1998a} states this should give [[\tscon{EDESTADDRREQ}]], not [[ENOTCONN]].

sendto_5 **<u>fail</u> as socket in an error state**

$$\begin{array}{l} FC(ifds, tid, (\text{RUN})_d, \\ \qquad s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := \uparrow e |\!\rangle) \\ \xrightarrow{tid \cdot \text{sendto}(s.fd, ips, data, nb)} FC(ifds, tid, (\text{RET}(\text{FAIL } e))_{\text{dsch}}, \\ \qquad s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := * |\!\rangle) \end{array}$$

F_context FC

Recall that by the [[socket-ok]] invariant if the error component is [[SOME e]] then the local port is not [[*]].

sendto_6 **<u>fail</u> as *data* size too big**

$$\begin{array}{l} FC(ifds, tid, (\text{RUN})_d, s) \\ \xrightarrow{tid \cdot \text{sendto}(s.fd, ips, data, nb)} FC(ifds, tid, (\text{RET}(\text{FAIL EMSGSIZE}))_{\text{dsch}}, s \text{ with } ps_1 := ps_1') \end{array}$$

F_context FC $\wedge$
string_size $data$ > UDPpayloadMax $\wedge$
$ps_1' \in \{s.ps_1\} \cup (\textbf{image}(\uparrow))(\text{autobind}(s.ps_1, FC)))$

In Linux, [[sendto_5]] takes precedence over this, so [[EMSGSIZE]] cannot happen in an error state. In a normal state, it does not set the error flag. What happens on a [[Sendto2]] if an error flag is set is open.
Note that we let [[ps1'=ps1]] nondeterministically, as the size check may be before or after the autobind. On Linux this is unnecessary as the autobind is performed before the [[EMSGSIZE]] check.
Note that we nondeterministically allow the message size check to fail either on entry to [[Sendto2]] or on exit.

sendto_7 **fail** **as there are no ephemeral ports left**

$$FC(\mathit{ifds}, \mathit{tid}, (\mathrm{R{\scriptsize UN}})_d, \mathrm{S{\scriptsize OCK}}(\mathit{fd}, *, *, *, *, *, f, \mathit{mq}))$$
$$\xrightarrow{\mathit{tid}\cdot\mathrm{sendto}(\mathit{fd},\mathit{ips},\mathit{data},\mathit{nb})} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{R{\scriptsize ET}}(\mathrm{FAIL}\ e))_{\mathrm{dsch}}, s \text{ with } \mathit{fd} := \mathit{fd})$$

F_context FC $\wedge$
(autobind$(*, FC) = \emptyset$) $\wedge$
$e \in$ ephemeralErrors $\wedge$
socket_ok $\mathit{ifds}(s \text{ with } \mathit{fd} := \mathit{fd})$

sendto_8 **slowsucceed** **[urgent]**

$$h \text{ with } \langle\!| \mathit{ts} := \mathbf{fupdate}\ \mathit{ts}\ (\mathit{tid}, (\mathrm{S{\scriptsize ENDTO}2}(s.\mathit{fd}, \mathit{ips}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{data}))_d); s := SC(s \text{ with } \mathit{es} := *) |\!\rangle$$
$$\xrightarrow{\tau} h \text{ with } \langle\!| \mathit{ts} := \mathit{ts} \oplus (\mathit{tid} \mapsto (\mathrm{R{\scriptsize ET}}(\mathrm{OK}()))_{\mathrm{dsch}}); s := SC(s \text{ with } \mathit{es} := *); \mathit{oq} := \mathit{oq}'; \mathit{oqf} := \mathit{oqf}' |\!\rangle$$

socklist_context SC $\wedge$
$((\mathit{oq}', \mathit{oqf}', \mathbf{T}) \in \mathrm{dosend}(h.\mathit{ifds}, (\mathit{ips}, \mathit{data}), (s.\mathit{is}_1, s.\mathit{ps}_1, s.\mathit{is}_2, s.\mathit{ps}_2), h.\mathit{oq}, h.\mathit{oqf}) \wedge$
$((\mathit{ips} \neq *) \vee (s.\mathit{is}_2 \neq *)) \vee$
$(\mathit{oq}', \mathit{oqf}', \mathbf{T}) \in \mathrm{dosend}(h.\mathit{ifds}, (\mathit{ips}, \mathit{data}), (\mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2), h.\mathit{oq}, h.\mathit{oqf}) \wedge$
$((\mathit{ips} \neq *) \vee (\mathit{is}_2 \neq *))) \wedge$
string_size $\mathit{data} \leq$ UDPpayloadMax

Note that the [[Sendto2]] state can only be entered by the [[sendto_2]] rule. Unfortunately, this does not guarantee that the socket's [[is2]] and [[ps2]] will be present, because a call to [[disconnect]] may have removed them. We allow for a successful send to occur based on the values in the socket when the call was made however.

sendto_9 **slowfail** **[urgent] as socket has entered an error state**

$$FC(\mathit{ifds}, \mathit{tid}, (\mathrm{S{\scriptsize ENDTO}2}(s.\mathit{fd}, \mathit{ips}, \mathit{data}))_d, s \text{ with } \mathit{es} := \uparrow e)$$
$$\xrightarrow{\tau} FC(\mathit{ifds}, \mathit{tid}, (\mathrm{R{\scriptsize ET}}(\mathrm{FAIL}\ e))_{\mathrm{dsch}}, s \text{ with } \mathit{es} := *)$$

F_context FC

sendto_10 **slowfail** **[urgent] as *data* size too big or socket now disconnnected**

$$
\begin{array}{l}
h \text{ with } \langle\!|\, ts := \mathbf{fupdate}\ ts \\
\qquad\qquad\qquad (tid, (\textsc{Sendto2}(fd, ips, is_1, ps_1, is_2, ps_2, data))_d); \\
\qquad\quad s := SC(s \text{ with } es := *) \,|\!\rangle \\
\xrightarrow{\tau} h \text{ with } \langle\!|\, ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}); \\
\qquad\quad s := SC(s \text{ with } es := *) \,|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{socklist\_context } SC \wedge \\
(\exists oq'\, oqf'. \\
\quad (oq', oqf', \mathbf{T}) \in \text{dosend}(h.ifds, (ips, data), \\
\qquad\qquad (is_1, ps_1, is_2, ps_2), \\
\qquad\qquad h.oq, h.oqf)) \wedge \\
((\text{string\_size } data > \text{UDPpayloadMax} \wedge (e = \text{EMSGSIZE}) \vee \\
\quad (s.is_2 = *) \wedge (ips = *) \wedge (e = \text{ENOTCONN})))
\end{array}
$$

The condition ensures it fires only when we try to put the message onto the outqueue, modelling the discovery at that point that the message is too large, or that the socket has become disconnected. (This rule nondeterministically competes with *sendto_6*, which would catch an outsize message earlier, and with *sendto_8*, which would allow the delivery of a message on a now-disconnected socket based on the values of $s.is_2$ and $s.ps_2$ when the call was made.
This rule leaves the socket state unchanged. One might wish to allow it to have been disturbed, nondeterministically.

recvfrom_1 **succeed**

$$
\begin{array}{l}
\qquad FC(ifds, \\
\qquad\quad tid, (\textsc{Run})_d, \\
\qquad\quad s \text{ with } \langle\!|\, ps_1 := \uparrow p_1; es := *; \\
\qquad\qquad mq := (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid) :: mq \,|\!\rangle) \\
\xrightarrow{tid \cdot \text{recvfrom}(s.fd, nb)} FC(ifds, tid, (\textsc{Ret}(\text{OK}(i_3, ps_3, data)))_{\text{dsch}}, \\
\qquad\quad s \text{ with } \langle\!|\, ps_1 := \uparrow p_1; es := *; mq := mq \,|\!\rangle)
\end{array}
$$

$$\text{F\_context } FC$$

recvfrom_2 **enter2** **entering Recvfrom2 state**

$$
\begin{array}{l}
\qquad FC(ifds, tid, (\textsc{Run})_d, s \text{ with } \langle\!|\, es := *; mq := [\,] \,|\!\rangle) \\
\xrightarrow{tid \cdot \text{recvfrom}(s.fd, \mathbf{F})} FC(ifds, tid, (\textsc{Recvfrom2}\, s.fd)_\infty, \\
\qquad\quad s \text{ with } \langle\!|\, es := *; mq := [\,]; ps_1 := \uparrow p_1' \,|\!\rangle)
\end{array}
$$

$$
\begin{array}{l}
\text{F\_context } FC \wedge \\
p_1' \in \text{autobind}(s.ps_1, FC)
\end{array}
$$

The autobinding can be observed by guessing the next ephemeral port.

recvfrom_3 **fail** **as would block**

$$
\begin{array}{l}
FC(ifds, tid, (\textsc{Run})_d, s \text{ with } \langle\!| es := *; mq := [\,] |\!\rangle) \\
\xrightarrow{tid\cdot\text{recvfrom}(s.fd, \mathbf{T})} FC(ifds, tid, (\textsc{Ret}(\text{FAIL EAGAIN}))_{\text{dsch}}, \\
\qquad s \text{ with } \langle\!| es := *; mq := [\,]; ps_1 := \uparrow p_1' |\!\rangle)
\end{array}
$$

F_context FC $\wedge$
$p_1' \in \text{autobind}(s.ps_1, FC)$

Note that in Linux [[\tscon{EWOULDBLOCK}]] and [[EAGAIN]] are aliased.

recvfrom_4 **fail** **as socket in an error state**

$$
\begin{array}{l}
FC(ifds, tid, (\textsc{Run})_d, s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := \uparrow e |\!\rangle) \\
\xrightarrow{tid\cdot\text{recvfrom}(s.fd, nb)} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := * |\!\rangle)
\end{array}
$$

F_context FC

recvfrom_5 **fail** **as there are no ephemeral ports left**

$$
\begin{array}{l}
FC(ifds, tid, (\textsc{Run})_d, \textsc{Sock}(fd, *, *, *, *, *, f, [\,])) \\
\xrightarrow{tid\cdot\text{recvfrom}(fd, nb)} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, s \text{ with } fd := fd)
\end{array}
$$

F_context FC $\wedge$
$(\text{autobind}(*, FC) = \emptyset)$ $\wedge$
$e \in \text{ephemeralErrors}$ $\wedge$
socket_ok $ifds(s \text{ with } fd := fd)$

See remark for [[connect_3]].

recvfrom_6 **slowsucceed** **[urgent]**

$$
\begin{array}{l}
FC(ifds, tid, (\textsc{Recvfrom2}\, s.fd)_d, \\
\qquad s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := *; \\
\qquad\qquad mq := (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid) :: mq |\!\rangle) \\
\xrightarrow{\tau} FC(ifds, tid, (\textsc{Ret}(\text{OK}(i_3, ps_3, data)))_{\text{dsch}}, \\
\qquad s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := *; mq := mq |\!\rangle)
\end{array}
$$

F_context FC

Note that the [[Recvfrom2]] state can only be entered by the [[recvfrom_2]] rule, so we have [[ps1= SOME p1]] in the [[host-ok]] invariant.

recvfrom_7 **slowfail** **[urgent] as socket has entered an error state**

$$
\begin{array}{l}
FC(ifds, tid, (\textsc{Recvfrom2}\, s.fd)_d, s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := \uparrow e |\!\rangle) \\
\xrightarrow{\tau} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, s \text{ with } \langle\!| ps_1 := \uparrow p_1; es := * |\!\rangle)
\end{array}
$$

F_context FC

close_1 **succeed**

$$h \text{ with } \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d); \; s := s_1 \mathbin{@} (s :: s_2) |\!\rangle$$

$$\xrightarrow{tid \cdot \text{close}\, s.fd} h \text{ with } \langle\!| ts := \text{zombify } s.fd\, o\_f \; ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{dsch}}); \; s := s_1 \mathbin{@} s_2 |\!\rangle$$

T

The zombify function takes Sendto2 and Recvfrom2 threadstates involving $s.fd$ to the Zombie threadstate. Select2 threadstates involving $s.fd$ have that descriptor removed from the lists of file-descriptors maintained by that state. Note that we are unsure about this behaviour. Initial experiments using MiniCaml threads suggest that blocked RECVFROM2 threads remain blocked in this situation (and don't raise an error, as one might expect). Further experiments, particularly on blocked SENDTO2 states are required.

select_1 **enter2** **entering Select2 state**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d)$$

$$\xrightarrow{tid \cdot \text{select}(readseq, writeseq, tms)} h \text{ with } ts := ts \oplus (tid \mapsto (\text{SELECT2}(readseq, writeseq))_{d'})$$

$\textbf{list\_to\_set}(readseq \mathbin{@} writeseq) \subseteq \text{sockfds}\, h.s \wedge$
$(\forall i.(tms = \uparrow i) \implies 0 \leq i) \wedge$
$(d' = \textbf{case}\, tms\, \textbf{of}$
$\quad * \to \infty \parallel$
$\quad \uparrow i \to \text{time}(\text{real\_of\_int}\; i / 1000000))$

select_2 **fail** **as timeout negative**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d)$$

$$\xrightarrow{tid \cdot \text{select}(readseq, writeseq, \uparrow tm)} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{dsch}})$$

$tm < 0$

select_3 **slowsucceed** **[urgent] with at least one** fd **ready**

$$
\begin{array}{l}
\quad h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Select2}(readseq, writeseq))_d) \\
\xrightarrow{\tau} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}(readseq', writeseq')))_{\mathrm{dsch}})
\end{array}
$$

$$
\begin{array}{l}
(readseq' = \textbf{filter} \\
\qquad\qquad (\lambda fd.\exists s.\, \textbf{mem}\ s\ h.s \wedge (s.fd = fd) \wedge \\
\qquad\qquad\qquad ((s.mq \neq [\,]) \vee (s.es \neq *))) \\
\qquad\qquad readseq) \wedge \\
(writeseq' = \textbf{filter} \\
\qquad\qquad (\lambda fd.\exists s.\, \textbf{mem}\ s\ h.s \wedge (s.fd = fd) \wedge \\
\qquad\qquad\qquad (h.oqf = \mathbf{F})) \\
\qquad\qquad writeseq) \wedge \\
((readseq' \neq [\,]) \vee (writeseq' \neq [\,]))
\end{array}
$$

select_4 **slowsucceed** **timeout reached**

$$
\begin{array}{l}
\quad h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Select2}(readseq, writeseq))_0) \\
\xrightarrow{\tau} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}(readseq', writeseq')))_{\mathrm{dsch}})
\end{array}
$$

$$
\begin{array}{l}
(readseq' = \textbf{filter}(\lambda fd.\exists s.\, \textbf{mem}\ s\ h.s \wedge (s.fd = fd) \wedge \\
\qquad\qquad\qquad ((s.mq \neq [\,]) \vee (s.es \neq *))) \\
\qquad\qquad readseq) \wedge \\
(writeseq' = \textbf{filter}(\lambda fd.\exists s.\, \textbf{mem}\ s\ h.s \wedge (s.fd = fd) \wedge \\
\qquad\qquad\qquad (h.oqf = \mathbf{F})) \\
\qquad\qquad writeseq) \wedge \\
(readseq' = [\,]) \wedge (writeseq' = [\,])
\end{array}
$$

delay_1 **enter2** **entering Delay2 state**

$$
\begin{array}{l}
\qquad\qquad h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \\
\xrightarrow{tid \cdot \mathsf{delay}\, tm} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Delay2})_{d'})
\end{array}
$$

$$
\begin{array}{l}
(tm \geq 0) \wedge \\
(d' = \mathrm{time}(\mathrm{real\_of\_int}\ tm/1000000))
\end{array}
$$

delay_2 **fail** **as timeout negative**

$$
\begin{array}{l}
\qquad\qquad h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \\
\xrightarrow{tid \cdot \mathsf{delay}\, tm} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{FAIL\ EINVAL}))_{\mathrm{dsch}})
\end{array}
$$

$$tm < 0$$

delay_3 **slowsucceed** **when timer reaches zero**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{DELAY2})_0) \\ \xrightarrow{\tau} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{dsch}}) \end{array}$$

T

getifaddrs_1 **succeed**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d) \\ \xrightarrow{tid \cdot \text{getifaddrs}()} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}\, \mathit{iflist}))_{\text{dsch}}) \end{array}$$

$$\begin{array}{l} \mathit{iflist} \in \text{orderings}\, \mathit{ifset} \wedge \\ (\mathit{ifset} = \{(\mathit{hifd}.\mathit{ifid}, \mathit{hifd}.\mathit{primary}, \mathit{ipslist}, \mathit{hifd}.\mathit{netmask}) | \\ \qquad \mathit{hifd} \in h.\mathit{ifds} \wedge \\ \qquad \mathit{ipslist} \in \text{orderings}\, \mathit{hifd}.\mathit{ipset}\}) \end{array}$$

Note this returns the interfaces in any arbitrary order each time; likewise the aliases within each interface. If one wanted otherwise, [[ifds]] and [[iset]] should be lists.

console_print_1 **enter2** **enter the Print2 state**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d) \\ \xrightarrow{tid \cdot \text{print\_endline\_flush}\, v} h \text{ with } ts := ts \oplus (tid \mapsto (\text{PRINT2}v)_{\text{dsch}}) \end{array}$$

T

console_print_2 **misc** **print the string on console**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{PRINT2}v)_d) \\ \xrightarrow{\overline{\text{CONSOLE } v}} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}()))_{\text{dsch}}) \end{array}$$

T

convert_port_1 **succeed**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d) \\ \xrightarrow{tid \cdot \text{port\_of\_int}\, v} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{PORT}(\mathbf{num}\, v))))_{\text{dsch}}) \end{array}$$

$$1 \leq v \wedge v \leq 65535$$

convert_port_2 **fail**

$$\begin{array}{l} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RUN})_d) \\ \xrightarrow{tid \cdot \text{port\_of\_int}\, v} h \text{ with } ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINVAL}))_{\text{dsch}}) \end{array}$$

$$v < 1 \vee v > 65535$$

convert_ip_1 **succeed**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot \text{ip\_of\_string}\, v} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK}\, i))_{\text{dsch}})$$

valid_ip_string v $\wedge$
$(i = \text{ipstr\_to\_ip}\ v)$

convert_ip_2 **fail**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot \text{ip\_of\_string}\, v} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINVAL}))_{\text{dsch}})$$

$\neg$ valid_ip_string v

notsockfd_1 **fail** select **as some** fd **not a socket file descriptor**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot \text{select}(readseq, writeseq, tms)} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EBADF}))_{\text{dsch}})$$

$\exists fd.\, \mathbf{mem}\ fd\ (readseq\ @\ writeseq) \wedge fd \notin \text{sockfds}\, h.s$

Error is 'Bad file number'.

notsockfd_2 **fail** OP **as** fd **not a socket file descriptor**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot opn} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}})$$

$opn \in \text{OP}\, fd \wedge$
$fd \notin \text{sockfds}\, h.s \wedge$
$e \in \{\text{ENOTSOCK}; \text{EBADF}\}$

The errors are 'Socket operation on non-socket' and 'Bad file number'.

intr_1 **slowfail as system call interrupted**

$$h \text{ with } ts := ts \oplus (tid \mapsto (opn)_d)$$
$$\xrightarrow{\tau} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINTR}))_{\text{dsch}})$$

$(\exists v.opn = \textsc{Sendto2}\, v) \vee$
$(\exists v.opn = \textsc{Recvfrom2}\, v) \vee$
$(\exists v.opn = \textsc{Select2}\, v) \vee$
$(\exists v.opn = \textsc{Print2}\, v) \vee$
$(opn = \textsc{Delay2}) \vee$
$(opn = \textsc{Zombie})$

(need to add explanation here)

$badmem\_1$ **badfail** socket **as no space**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \xrightarrow{tid\cdot\text{socket()}} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}})$$

$e \in \{\text{ENOMEM}; \text{ENOBUFS}\}$

The errors are (no space to allocate) and (no space for buffers) respectively.

$badmem\_2$ **badfail** OP **as no space**

$$FC(ifds, tid, (\textsc{Run})_d, s) \xrightarrow{tid\cdot opn} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, s)$$

F_context FC $\wedge$
$opn \in \text{OP}(s.fd)$ $\wedge$
$e \in \{\text{ENOMEM}; \text{ENOBUFS}\}$

The errors are (no space to allocate) and (no space for buffers) respectively.

$badmem\_3$ **badfail** OP2 **as no space**

$$FC(ifds, tid, (t)_d, s) \xrightarrow{\tau} FC(ifds, tid, (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}}, s)$$

F_context FC $\wedge$
$t \in \text{OP2}(s.fd)$ $\wedge$
$e \in \{\text{ENOMEM}; \text{ENOBUFS}\}$

The errors are (no space to allocate) and (no space for buffers) respectively.

$badmem\_4$ **badfail** **Select2 as no space**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Select2}v)_d) \xrightarrow{\tau} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL } e))_{\text{dsch}})$$

$e \in \{\text{ENOMEM}; \text{ENOBUFS}\}$

The errors are 'Out of memory' and 'No buffer space available'.

$exit\_1$ **exit**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \xrightarrow{tid\cdot\text{exit()}} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Exit})_{\text{dsch}})$$

T

exit_2 **exit**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Exit})_d)$$
$$\xrightarrow{\text{EXIT}} h \text{ with } \langle\!\langle ts := \emptyset;\ s := [\,] \rangle\!\rangle$$

T

ret_1 **misc** **return value v from fast system call to thread**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}\, v)_d)$$
$$\xrightarrow{\overline{tid \cdot v}} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_\infty)$$

T

create_1 **succeed** **create new thread**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot \text{create}()} h \text{ with } ts := \mathbf{fupdate}(ts \oplus (tid \mapsto (\textsc{Ret}(\text{OK}\, tid'))_{\text{dsch}}))$$
$$(tid', (\textsc{Run})_\infty)$$

$tid' \notin (\{tid\} \cup \mathbf{fdom}\, ts)$

create_2 **badfail** **no space for new thread**

$$h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Run})_d)$$
$$\xrightarrow{tid \cdot \text{create}()} h \text{ with } ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EAGAIN}))_{\text{dsch}})$$

T

The man page for pthread_create explains that EAGAIN is returned if there aren't enough system resources, or if there are more threads than PTHREAD _THREADS_MAX.

delivery_out_1 **misc** **put UDP or ICMP to the network from oq**

$$h$$
$$\xrightarrow{\overline{\text{IP}(i_1, i_2, body)}} h \text{ with } \langle\!\langle oq := oq';\ oqf := oqf' \rangle\!\rangle$$

$(\text{IP}(i_1, i_2, body), oq', oqf') \in \text{dequeue}(h.oq, h.oqf) \wedge$
$i_2 \notin \text{LOOPBACK} \cup \text{MARTIAN} \wedge$
$i_1 \notin \text{MARTIAN}$

dequeue resets the oq' timer to ∞ if the queue is empty, or to doq otherwise. Note that dequeue gives an empty set of possible outcomes if $oq = [\,]$.

delivery_out_martian **misc** **discard martian IP from** *oq*

$$h \xrightarrow{\tau} h \text{ with } \langle\!| oq := oq'; oqf := oqf' |\!\rangle$$

$\exists i_1\, i_2\, body.(\mathrm{IP}(i_1, i_2, body), oq', oqf') \in \mathrm{dequeue}(h.oq, h.oqf) \wedge$
$\quad i_2 \in \mathrm{MARTIAN}$

This rule is speculative. Note that it applies to UDP and to ICMP, and to loopback destinations.

delivery_in_udp_1 **misc** **get UDP from network and deliver it to a matching socket**

$$h \text{ with } s := SC\,s \xrightarrow{\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data))} h \text{ with } s := SC(s \text{ with } mq := s.mq \mathbin{@} [(\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), ifd.ifid)])$$

$\mathrm{socklist\_context}\ SC \wedge$
$s \in \mathrm{lookup}(SC\,s)(i_3, ps_3, i_4, ps_4) \wedge$
$ifd \in h.ifds \wedge$
$i_4 \in ifd.ipset \wedge$
$i_4 \notin \mathrm{LOOPBACK} \wedge$
$i_3 \notin \mathrm{MARTIAN} \cup \mathrm{LOOPBACK}$

delivery_in_udp_2 **misc** **get UDP from network but generate ICMP, as no matching socket**

$$h \xrightarrow{\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data))} h \text{ with } \langle\!| oq := oq'; oqf := oqf' |\!\rangle$$

$i_4 \in h.ifds \wedge$
$(\mathrm{lookup}\, h.s(i_3, ps_3, i_4, ps_4) = \emptyset) \wedge$
$(oq', oqf', ok) \in$
$\quad (\{(h.oq, h.oqf, \mathbf{T})\} \cup$
$\quad \mathrm{enqueue}(\mathrm{IP}(i_4, i_3, \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4)),$
$\qquad h.oq, h.oqf)) \wedge$
$i_4 \notin \mathrm{LOOPBACK} \wedge$
$i_3 \notin \mathrm{MARTIAN} \cup \mathrm{LOOPBACK}$

Note that ICMP generation is unreliable; the Linux kernel has per-type-of-ICMP rate-limiting to control denial-of-service attacks. We take a nondeterministic approximation. Note also that the ICMP is dropped if the outqueue is full ([[ok=false]]). \mignore{In the real world, there is no socket to charge this to... so who says "no room to allocate skb"?}

$delivery\_in\_martian\_3$ **misc** **get martian IP from network and discard**

$$h \xrightarrow{\mathrm{IP}(i_3, i_4, body)} h$$

$i_4 \in h.ifds \wedge$
$i_4 \notin \mathrm{LOOPBACK} \wedge$
$(\mathrm{loopback}(\mathrm{IP}(i_3, i_4, body)) \vee \mathrm{martian}(\mathrm{IP}(i_3, i_4, body)))$

This is a garbage collection rule to pull martian UDP or ICMP off of the network and discard them.

$delivery\_in\_icmp\_1$ **misc** **get ICMP from the network, setting error in a matching socket**

$$h \text{ with } s := SC\,s \xrightarrow{\mathrm{IP}(i_4', i_3', body)} h \text{ with } s := SC(s \text{ with } es := es')$$

$\mathrm{socklist\_context}\ SC \wedge$
$((body = \mathrm{ICMP\_HOST\_UNREACH}(i_3, ps_3, i_4, ps_4)) \vee$
$\quad (body = \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4))) \wedge$
$s \in \mathrm{lookup}(SC\,s)(i_3, ps_3, i_4, ps_4) \wedge$
$i_3' \in h.ifds \wedge$
$\neg(\mathrm{loopback}(\mathrm{IP}(i_4', i_3', body)) \vee \mathrm{martian}(\mathrm{IP}(i_4', i_3', body))) \wedge$
$(es' = \textbf{if } (s.is_2 \neq *) \vee \neg s.f.bsdcompat$
$\qquad \textbf{then } \uparrow\mathrm{ECONNREFUSED}$
$\qquad \textbf{else } s.es)$

As we are not sure which destination ([[i3]], [[i3']]) address the matching is done on, we add the [[LOOPBACK]] check to both to make the rule sensible. Should (re?)check the source.

$delivery\_in\_icmp\_2$ **misc** **get ICMP from the network and drop as no matching socket**

$$h \xrightarrow{\mathrm{IP}(i_4', i_3', body)} h$$

$i_3' \in h.ifds \wedge$
$((body = \mathrm{ICMP\_HOST\_UNREACH}(i_3, ps_3, i_4, ps_4)) \vee$
$\quad (body = \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4))) \wedge$
$(\mathrm{lookup}\ h.s(i_3, ps_3, i_4, ps_4) = \emptyset) \wedge$
$\neg(\mathrm{loopback}(\mathrm{IP}(i_4', i_3', body)) \vee \mathrm{martian}(\mathrm{IP}(i_4', i_3', body)))$

$\textit{delivery\_loopback\_udp\_1}$ **<u>misc</u> get loopback UDP from oq and deliver it to a matching socket**

$$
\begin{aligned}
& h \text{ with } s := SC\,s \\
\xrightarrow{\tau}\ & h \text{ with } \langle\!| s := SC(s \text{ with } mq := s.mq \mathbin{@} [(\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), \mathit{ifid})]); \\
& \qquad\qquad oq := oq';\ oqf := oqf' |\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{socklist\_context}\ SC \wedge \\
& (\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), oq', oqf') \in \mathrm{dequeue}(h.oq, h.oqf) \wedge \\
& i_4 \in \mathrm{LOOPBACK} \wedge \\
& s \in \mathrm{lookup}(SC\,s)(i_3, ps_3, i_4, ps_4) \wedge \\
& \langle\!| \mathit{ifid} := \mathit{ifid};\ ipset := \mathrm{LOOPBACK};\ primary := \mathrm{localhost}; \\
& \quad netmask := \mathrm{NETMASK}(255 * (2 ** 24)) |\!\rangle \in h.\mathit{ifds}
\end{aligned}
$$

[[dequeue]] resets the [[oq']] timer to [[time_infty]] if the queue is empty, or to [[doq]] otherwise.
Note that [[\{i3,i4\}\cap MARTIAN = \emptyset]] is guaranteed by invariants.

$\textit{delivery\_loopback\_udp\_2}$ **<u>misc</u> get a loopback UDP from oq but generate loopback ICMP, as no matching socket**

$$
\begin{aligned}
& h \\
\xrightarrow{\tau}\ & h \text{ with } \langle\!| oq := oq'';\ oqf := oqf'' |\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
& (\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), oq', oqf') \in \mathrm{dequeue}(h.oq, h.oqf) \wedge \\
& i_4 \in \mathrm{LOOPBACK} \wedge \\
& (\mathrm{lookup}\ h.s(i_3, ps_3, i_4, ps_4) = \emptyset) \wedge \\
& (oq'', oqf'', ok) \in \\
& \mathrm{enqueue}(\mathrm{IP}(i_4, i_3, \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4)), oq', oqf')
\end{aligned}
$$

ICMPs are dropped if the outqueue is full (the [[ok]] flag is ignored). Note that [[\{i3,i4\}\cap MARTIAN = \emptyset]] is guaranteed by invariants.

$\textit{delivery\_loopback\_icmp\_1}$ **<u>misc</u> get a loopback ICMP from oq and deliver (by setting es) to a matching socket**

$$
\begin{aligned}
& h \text{ with } s := SC\,s \\
\xrightarrow{\tau}\ & h \text{ with } \langle\!| s := SC(s \text{ with } es := es');\ oq := oq';\ oqf := oqf' |\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{socklist\_context}\ SC \wedge \\
& (\mathrm{IP}(i_4', i_3', \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4)), oq', oqf') \in \mathrm{dequeue}(h.oq, h.oqf) \wedge \\
& i_3' \in \mathrm{LOOPBACK} \wedge \\
& s \in \mathrm{lookup}(SC\,s)(i_4, ps_4, i_3, ps_3) \wedge \\
& (es' = \textbf{if } (s.is_2 \neq *) \vee \neg s.f.bsdcompat \textbf{ then } \uparrow\mathrm{ECONNREFUSED} \\
& \qquad\quad \textbf{else } s.es)
\end{aligned}
$$

Note that [[\{i4',i3',i3,i4\}\cap MARTIAN = \emptyset]] is guaranteed by invariants.

delivery_loopback_icmp_2 **misc** **get a loopback ICMP from** oq **and drop, as no socket matches**

$$h$$
$$\xrightarrow{\tau} h \text{ with } \langle\!| oq := oq'; oqf := oqf' |\!\rangle$$

$$(\mathrm{IP}(i_4', i_3', \mathrm{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4)), oq', oqf') \in$$
$$\quad \mathrm{dequeue}(h.oq, h.oqf) \wedge$$
$$i_3' \in \mathrm{LOOPBACK} \wedge$$
$$(\mathrm{lookup}\, h.s(i_4, ps_4, i_3, ps_3) = \emptyset)$$

Note that [[\{i4',i3'\}\cap MARTIAN = \emptyset]] is guaranteed by invariants.

epsilon_1 **misc** **time passage**

$$h$$
$$\xrightarrow{d} h'$$

$$d > 0 \wedge$$
$$(\mathrm{Time\_Pass}\, d h = \uparrow h')$$