UC Irvine
ICS Technical Reports

Title
Specification of initial connection handling in TCP using structured Petri nets

Permalink
https://escholarship.org/uc/item/96k585bm

Author
Rose, Marshall T.

Publication Date
1984-03-06

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

699
C3
no. 219

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Specification of Initial Connection Handling in TCP using structured Petri nets

Marshall T. Rose
Department of Information and Computer Science
University of California, Irvine
Wed Aug 24 10:20:44 1983
Revised: Tue Mar 6 20:18:51 1984

Computer Mail: MRose@UCI

Technical Report Number 219

ABSTRACT

This paper uses structured Petri nets to specify how connection establishment is handled by the DoD Transmission Control Protocol. The purpose of this paper is to demonstrate an alternate specification technique by examining its application to a portion of a protocol of reasonable complexity.

Initially we briefly present the semantics of structured Petri nets. Following this, a terse discussion of the problems of establishing connections in a network takes place. This discussion centers on the use of the three-way handshake, which is is used by TCP, as a solution for many of these problems. Finally, the specification of the three-way handshake used in TCP is made. The specification is presented in three sections: first, a general set of notes concerning the nature of this particular specification is discussed; second, the data definitions of the specification are given; and, third, the actual nets themselves are presented.

This paper is condensed from a portion of the author's dissertation, which is still in preparation. In the interests of brevity, some components of the specification, such as retransmission handling, have been omitted. Interested readers should contact the author for a more detailed paper.

CONTENTS

Page

1. An Overview of structured Petri nets 1
   Topology 1
   Tokens 1
   Colors 1
   Enabling Predicates 2
   Firing Actions 2
   Transition Disciplines 2
   Named nets 3
   Named Subnets 4
   Graphical Conventions 4

2. Connection Establishment 5

3. Specification Notes 6

4. Specification Data Definitions 7

5. Specification Nets 9
   (N-1)-interface 9
   N-interface 11
   N-protocol 12
   N-protocol primitives 18

6. Evaluation 19

Conclusions 20

1. An Overview of structured Petri nets

Structured Petri nets are yet another augmentation to Petri nets[PETE77], and are inspired in part by numerical Petri nets[SYMO80A]. A more informal, intuitive description than the one presented here may be found in [MROSE83B]. The extensions found in the Structured Petri net model include dynamic scoping, hierarchy, and concurrent and recurrent invocation control.

Topology

A structured Petri net is a directed bi-partite graph populated with four types of nodes: *places*, which hold tokens; *transitions*, which absorb and produce tokens; *named nets*, which instantiate the execution of another structured Petri net; and *named subnets*, which denote the substitution of another structured Petri net. Arcs starting from a place or named net and leading to a transition are called *input* arcs, while arcs starting from a transition and leading to a place, named net, or subnet are called *output* arcs.

Tokens

Tokens traverse the net. Unlike the tokens used in numerical Petri nets, the tokens found in structured Petri nets are not unique. Instead, all tokens have a single attribute, a *color*. All tokens of the same color are indistinguishable from each other. A color is a conceptualism for a dynamically scoped environment. Colors are mapped to contour blocks, which represent a part of a private resource space for each N-peer executing the net. Hence, a contour is a binding between a particular N-peer and a given color.

Contour blocks contain bindings for variables and a static link, which is a pointer to a previous (scoping) contour. Using colorful tokens, structured Petri nets achieve a dynamic scoping mechanism to determine data access. When searching for a variable in the context of a particular token, the token's contour block is first examined. If the variable is not present, then the previous contour (found by using the static link) is examined. This process continues until either the binding for the variable is found or there is no previous contour (the variable is undefined in the context of the original token).

Colors

The color attribute of a token has an ordinal value from the set of all colors. Each N-peer has an associated color-generator, which produces new colors for the N-peer when the generator is *incremented*. The generator has a *current color value*, which is set to the color produced by the last increment operation. The color-generator is accessible by all structured Petri nets executing for a particular N-peer.

In addition to the colors that may be produced by the generator, there is a special color, the *blank* color. Tokens with the blank color differ from other tokens in one important way: *each blank token is unique*. The same variable within the context of several blank tokens may have different values, depending on which blank token is being used to delimit the context.

Enabling Predicates

Each transition has associated with it a enabling predicate, which specifies if that transition is permitted to fire. Only one transition in a structured Petri net may fire at a given instant. If more than one transition is enabled, then a non-deterministic choice is made as to which transition is actually permitted to fire.

A transition's enabling predicate composes a single, possibly very complex, boolean expression. The enabling predicate may reference the number of tokens present in the places leading to the transition and the their distribution on the input arcs for the transition. Only tokens with the same color are considered when the expression is evaluated, but tokens of different colors may be considered separately. If more than one color of tokens is able to satisfy the enabling predicate, a non-deterministic choice is made as to which color is permitted to satisfy the conditions. These tokens, which are chosen to satisfy the enabling predicate are known as *eligible* tokens. In addition to considering the marking of the net, the enabling predicate is allowed to reference variables in the context of the eligible tokens. No memory, other than those variables considered in the context of the eligible tokens may be referenced by the enabling predicate. Further, the testing of the enabling predicate must not promote any sort of change in state.

Firing Actions

Each transition has associated with it a set of firing actions. Although several activities occur when a transition fires, and a degree of ordered and sequential operation is enforced during firing, the totality of a transition firing is considered to be atomic. Once a transition begins to fire, no other transition is considered to be enabled.

A transition's firing actions compose a set of operations that perform four tasks: first, the eligible tokens are removed from their input places, and a *selection rule* is consulted to determine to the number of tokens (zero or more) that are to be placed on each output arc; second, these tokens are *introduced* onto the selected output arcs, with possibly a new context as indicated by a *construction rule*; third, the introduced tokens are modified in the firing context, according to a set of *manipulation rules*; and, finally, the eligible tokens are removed from the input arcs, and the introduced tokens are placed on the output places.

Transition Disciplines

There are five types of transitions used in structured Petri nets: *normal* transitions, *boundary* transitions, *split* transitions, *entry* transitions, and *exit* transitions,

A normal transition is a transition which adheres exactly to the description of the enabling predicates and firing actions described above. Other transition disciplines, which modify these rules somewhat are described below. Although it is possible to combine the attributes of two or more of these types of transitions to achieve a hybrid, in the interests of clarity, the discussion that follows considers no such possibility.

A boundary transition is viewed as allowing the passing of information from one N-peer to another. Topologically, boundary transitions have a single output arc. The selection rule of a boundary transition is constrained to introduce exactly one token on this arc, and the construction rules are required to specify that this token have the blank color (in violation of the rules presented above). The manipulation rules are required to completely specify the variables which one N-peer communicates to the other. These variables have their values fully copied to the introduced token. As a result, any information passed from one N-peer to another is transferred to the latter's private resource space, ensuring that each N-peer's resource space is private.

A split transition is viewed as a local synchronization method for N-peers. Split transitions are allowed to violate a key tenet of the structured Petri net philosophy: they may consider different colors when evaluating their enabling predicates. This lapse is permitted under a very strict condition: Although different colors may be considered, only one color at a particular input arc is considered. Topologically, split transitions have the same number of input arcs and output arcs, and must have at least two of each. The enabling predicate for a split transition must require the presence of one token on each input place, the selection rule is constrained to introduce exactly one token on each output arc, and the construction rules must specify that each introduced token on a given output arc have the same color as the eligible token on the corresponding input arc.

An entry transition is viewed as the preparation that occurs prior to the call of a procedure or function in an algoritmic language. Topologically, entry transitions have a single output arc, which must be connected to an named net. The selection rule of a boundary transition is constrained to introduce exactly one token on this arc, and the construction rules are required to specify that the color generator be incremented and the resulting current color value of the generator be used as the color of the introduced token. The construction rules may specify variables to be defined in the new context, and the manipulation rules may initialize these variables. This achieves parameter-passing for the named net. Finally, the eligible tokens are removed and the introduced token is placed on the named net, which causes it to be instantiated. For reasons not explained herein, all parameters are required to be "passed by value," and the static link of the new context is made empty.

An exit transition is viewed as the clean-up that occur after the return of a call to a procedure or function. Topologically, exit transitions have a single input arc, which must be connected to an named net. The enabling predicate for an exit transition must require the presence of one token on its input place, and the construction rules must specify that the color of each introduced token is set to the color of the eligible token for the entry transition corresponding to this exit transition. As the reader might suspect, by using the context of the eligible token, the manipulation rules may make use of a possible return value.

Named nets

Named nets are references to other structured Petri nets. Topologically, they are similar to places, but are represented with a labeled square instead of circle. For

representational convenience, in addition to labeling a named net with its name, the parameters used when the named net is invoked may appear also within the labeled square, using the traditional parenthesised notation.

Named nets may have more than one entry place. In this case, each entry place is named, and the structured Petri net which instantiates the named net must specify (in the labeled square) which entry place is to be used. An named net needs only a single exit place, although it may contain more. This is for representational convenience only. Multiple exit places should be thought of as leading to a single, actual exit place. As a further enhancement, named nets may also be specified as "single entry, zero exit" nets. That is, once the net is instantiated, it never returns. In such cases, the named net need not feed an input arc to a transition, and may be thought of as a "terminal" node.

Named Subnets

Named subnets are references to other structured Petri nets; Topologically, they are similar to places, but are represented with a labeled square with a dashed outline instead of a circle.

Subnets are used as a representational convenience only. Subnets never connect to input arcs, only to output arcs. When a transition introduces a token for a named subnet, that token is placed at the entry place of the corresponding structured Petri net. Hence, subnets differ from named nets because no contour is saved or restored.

Graphical Conventions

The convention for drawing Structured Petri nets differs somewhat from the standard notation used for Petri nets. When drawing a transition, text appearing to the left of the transition is interpreted as the enabling predicate for the transition. Similarly, text appearing to the right of the transition is interpreted as the firing actions for the transition. Since the firing actions consist of three components (selection rules, construction rules, and manipulation rules), individual components are specified by prefixed them with their name or a short abbreviation (e.g., "sr:"). As a short-hand notation, if the text appearing to the right of a transition is not prefixed, then the text is presumed to be the manipulation rules for the transition.

By default, transitions use an AND input logic for eligible tokens and an AND output logic for introduced tokens. Unless otherwise noted, the context of the tokens introduced when a transition fires is identical to the context of the tokens which enabbled that transition. For entry transitions, of course, the color generator is incremented.

For those places which feed input arcs leading to more than one transition, a "twiddle" symbol (e.g., '$\sim$') may be used as the enabling predicate for one of the transitions. This is a short-hand expression meaning that none of the other transitions being fed by the place have their enabling predicates satisfied. Similarly, for those transitions with more than one output arc, a twiddle symbol may be used as the selection rule for one of the arcs. As expected, this is a short-hand expression meaning that none of the selection rules associated with the other output arcs introduce any tokens.

By clever use of these conventions, the graphical representations of structured Petri nets can be presented in a concise fashion and kept relatively free from clutter.

2. Connection Establishment

The problems found in providing reliable virtual-circuit service over a potentially unreliable packet-switched network are discussed in great detail in [SUNS78B]. A central problem in this area is ensuring that a connection established between two processes in the network becomes synchronized and remains so. Each peer of the N-protocol that provides reliable communications to these processes must agree as to the state of the connection, and then update that state as changes occur.

A connection may be viewed as traversing through a number of states at each end. These states trace the activity of the connection from non-existence to establishment, then to data transfer, and finally to closing. This paper focuses on the means by which two N-peers establish a connection. Initially, a connection may be viewed as being closed. When a process indicates that it is willing to accept a connection from another process, the connection for that process enters a listening state, where the process waits for another process to complete the connection. This is known as a "passive" open. After this point, another process may attempt to complete the connection, by performing an "active" open. Providing that all of the appropriate conditions are met, the connection progresses to the established state at both ends. Alternately, both processes may try to actively open the connection simultaneously, and the N-peers must be able to handle this situation correctly.

Connections occur between two sockets in the network. A socket is a pairing between the address for a host in the network and a local port number for that host. A connection may be uniquely specified by listing the two sockets participating in the connection, as a socket uniquely identifies the end-point of a particular conversation. Usually, for passive opens, a process requests a local port number and does not specify a foreign socket. In contrast, for active opens, a process requests both a local port number and a foreign socket.

Information to be sent from one process to another is first given to the local N-peer, which then encapsulates the information in a *segment*. In addition to containing the data to be transmitted, the segment contains control information for the use of the N-peers. For our purposes, we are interested in three control bits that can be found in the segments that the N-peers exchange. The SYN bit indicates that an N-peer is requesting initial synchronization. The ACK bit indicates that an N-peer is acknowledging a previously received segment. The RST bit indicates that an N-peer is demanding that the connection should be reset.

At some time, a segment containing a SYN arrives. When the foreign and local sockets specified in the segment match the socket pairing specified by a process' open, a connection begins. To synchronize the connection, initial sequence numbers are exchanged between the two peers. These sequence numbers impose an ordering on the data exchanged by

the peers. Selection of the initial sequence number is a tricky business, as one must take great care to ensure that segments floating about from instantiations of previous connections have sequence numbers outside the range of legitimate sequence numbers. Once both peers have selected an initial sequence number, informed the other peer of the number, and received an acknowledgement, the connection becomes established. If something goes amiss in the connection establishment, a peer sends a segment containing a RST to the other peer. This has the effect of removing all traces of the connection, with the appropriate information returned to the process associated with the peer receiving the RST. Other information may be exchanged during this synchronization, including process data (to be passed up only when the connection is fully established), but these considerations are not germane to the focus of this paper.

This method of synchronization is referred to as a "three-way" handshake, as the simplest case of its operation can be summarized as: Process A performs a passive open. Some time later, process B performs an active open. This results in process B's TCP choosing an initial sequence number and sending a segment containing a SYN and the sequence number to process A's TCP. Process A's TCP receives the segment, examines it, chooses an initial sequence number of its own, and responds by sending a segment containing a SYN, the sequence number, and an ACK of the incoming segment. Process B's TCP receives the segment, examines it, and decides that the connection is established. In addition, process B's TCP sends a segment acknowledging the incoming segment. Upon receipt of this segment, process A's TCP also decides that the connection is established.

The three-way handshake is able to successfully deal with a large number of variations and exceptions, including such events as simultaneous active opens, duplicate segments, segments from other instantiations, and half-open connections (which occur when one host crashes during the synchronization activity, and loses all knowledge of a connection). Hence, it is a good initial connection protocol for use by a reliable virtual-circuit service.

3. Specification Notes

This section provides a general explanation of the specification that follows.

First, it must be noted that the following sections do not provide a complete specification of TCP. Only those aspects of TCP that deal with connection establishment are examined. Further, some aspects of TCP that play a minor role in connection establishment are abstracted to avoid unnecessary detail. Components of TCP which are given little attention are: TCP options and option handling, urgent data handling, windowing, user time-outs, and precedence and security/compartment considerations. The specification presented pays varying (small) degrees of attention to these aspects of TCP. In contrast, [TCP], the TCP specification fully considers all of these issues in its discussion of initial connection establishment.

Second, our specification uses short mnemonics to represent error conditions which may be raised and given to the user process. Their meanings are:

REFUSED — the connect was refused by the foreign peer

RESET — the connection was reset by the foreign peer

ISCONN — the connection is already established

NOBUFS — insufficient resources to service request

NOPEER — no foreign socket specified in an active *open*() call

DENIED — the user process is not allowed to specify this type of *open*() call

Third, the underlying (N-1)-service is presumed to be the DoD Internet Protocol[IP]. The discussion of the (N-1)-interface describes the type of service expected.

4. Specification Data Definitions

This section describes the data structures used by the specification. Three major structures are explicitly used, the *tcb*, *ip_type*, and *segment_type* structures.

The *tcb* structure contains all of the state information for a connection. In addition to the local and foreign sockets, all sequencing and windowing information, precedence and security/compartment information, and so forth are all kept in this structure. In the specification that follows, a *tcb* completely contains all known information about a connection for an N-peer. A *tcb* is in fact its own contour, and as a result has its own unique color. In the next section, the reader should be able to appreciate the advantages and disadvantages that this interpretation permits. Unlike the specification in [TCP], the *tcb* does not have its state encoded as a variable, instead, structured Petri nets are used to denote the state of a *tcb*.

The *ip_type* structure is the data type passed to/from the (N-1)-layer. This structure specifies the source and destination addresses, precedence and security/compartment information, a protocol code (which for our purposes is always the code for TCP), a time-to-live value, and a segment to be communicated. The *ip_type* structure is used as a part of the (N-1)-interface definition.

The *segment_type* structure is passed between N-peers as their means of communication. The definition of the *segment_type* follows fairly closely the definition given in the TCP specification, with a few exceptions. Each segment has associated with it a length. This is not kept explicitly in the segment, but instead is calculated based upon the values of various components in the segment. In order to make this specification more clear, the length is treated as an explicit component of a *segment_type*. In addition, although the source and destination addresses are not present in each segment, they have been made explicit components of a *segment_type* as well. Similarly, pointers to the urgent, data, and options portions of the *segment_type* have been abstracted somewhat.

All of the major data structures used in the specification are presented in figure 1. Figure 1 presents these in a rough C-like syntax, and a complete description of the semantics of these structures is not presented here.

Data Definitions

```
struct  tcb {
        socket_type        lsock, fsock;
        precedence_type    prc;
        security_type      sec;
        boolean            active_open;
        sequence_type      iss, irs;
        window_type        wnd;
        s_wnd_type         snd;
        r_wnd_type         rcv;
        usr_sig_type       timeout;
        segment_type       msg, seg;          /* not state information */
        segment_queue      retransmit;
};
struct  ip_type {
        addr_type          saddr, daddr;
        protocol_type      proto;
        precedence_type    prc;
        security_type      sec;
        time_type          ttl;
        segment_type       data;
};
struct  segment_type {
        addr_type          saddr, daddr;      /* not actually in the segment */
        port_type          sport, dport;
        seg_flags_type     ctl;
        integer            len;               /* not actually in the segment */
        sequence_type      seq, ack;
        window_type        wnd;
        seg_ptr            up, data, options;
        integer            data_offset;
        checksum_type      cksum;
};
struct  s_wnd_type {
        sequence_type      una, nxt, up, wl1, wl2;
        window_type        wnd;
};
struct  r_wnd_type {
        sequence_type      nxt, up;
        window_type        wnd;
};
struct  socket_type {
        addr_type          addr;
        port_type          port;
};
struct  seg_flags_type {
        boolean            urg, ack, psh, rst, syn, fin;
};
```

Figure 1. Data Definitions

5. Specification Nets

This section presents the actual specification itself. A series of structured Petri nets are presented, along with additional explanatory text.

Some conventions are used in the drawing of these nets which should be noted. First, the graphical conventions presented earlier in this paper are followed. Our primary motivation for this is to reduce clutter and make the nets appear more readable. As expected, this results in the loss of some simplicity. For example, some of the transitions presented have more than one output arc. One of these arcs may lead to a named net, and another may not. This means that the transition also serves as an entry transition if the selection rule for the output arc feeding the named net produces a token. Hybrid constructs of this sort are meant as a convenient short-hand notation.

Second, the entire specification is not composed entirely of nets. Very often, the enabling predicate or firing actions of a transition may reference an external routine or predicate to provide a value. These functions are assumed to be called in the current context. An example of an enabling predicate so specified would be one that uses the expression *P/S okay*, which appears throughout the specification and means "Are the precedence and security/compartment conditions properly met?". An example of a routine appearing in the firing actions for a transition would be *new_iss*(), which generates a new initial sequence number for a connection.

(N-1)-interface

The (N-1)-interface specification is achieved through two nets, SNDPKT and RCVPKT.

The SNDPKT net (figure 2) loads the appropriate information for the (N-1)-layer into an *ip_type* structure and then forks control. One fork leads to the net's exit place. The other leads to a place which feeds one of two transitions. One transition consumes the eligible token and does not introduce a token on its output arc. This represents the packet being lost by the (N-1)-layer. The other transition is a boundary transition. The firing actions for this transition state that the data structure known as *packet* crosses into the (N-1)-media.

Recall that firing actions, while indivisible, execute in an ordered fashion. Hence, the manipulation rules, after loading each field of the *ip_type* structure, calls *do_chksum*() to calculate the TCP checksum for the segment and the (imaginary) IP header. This corresponds to an interesting nuance in the TCP specification that actually includes information not found in the TCP segment into the checksum stored in the segment. Note that the although the selection rules for the transition introduced a token on both output arcs, only one arc had construction rules and manipulation rules to execute. Since only one set of manipulation rules is present, the order of execution is unimportant.

The RCVPKT net (figure 3) is instantiated to accept information from the (N-1)-media. When a token enters the net, it waits for a split transition to be enabled. This transition will be enabled when a token is present on the other input arc which contains a

(N-1)-interface: send a packet

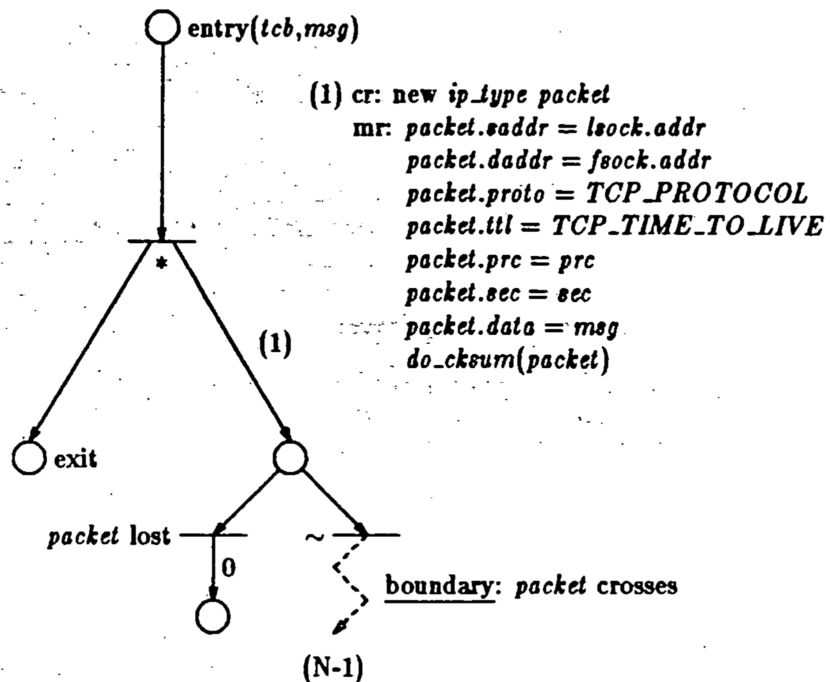


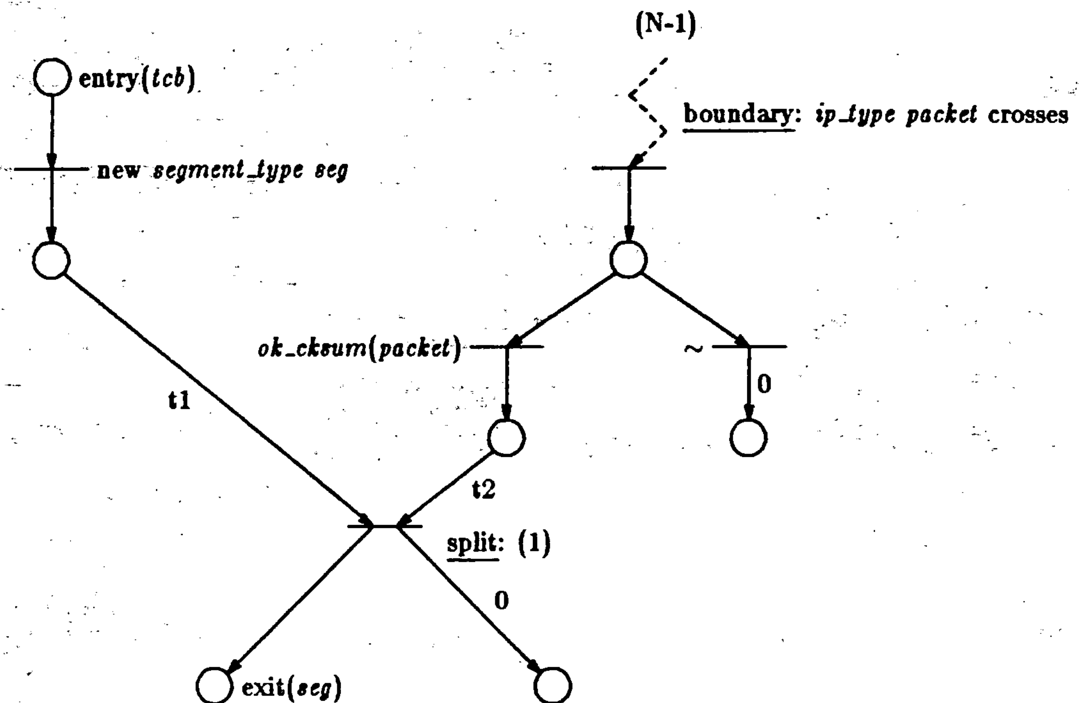
Figure 2. SNDPKT

packet for the connection represented by the entry token. That is, control will block until an incoming packet for the connection arrives at the other input place. Independently of this, whenever any packet arrives, an *ip_type* structure is given to the N-peer by the (N-1)-layer, and placed in a blank token by the boundary transition. The routine *ok_chksum*() is called to verify the checksum of the incoming packet (using the same algorithm which checks information in both the segment and the (imaginary) IP header). If the checksum found in the segment is incorrect, the blank token containing the *ip_type* structure is dropped. Otherwise, the token proceeds to a place to wait for the N-peer to request the next segment. When this split transition is enabled, the firing actions specify that the incoming packet is copied into the entry token's variable *packet*, and that $seq = packet.data$ in the context of this token. After firing, as with all split transitions, control forks. The fork corresponding to the entry token exits, which returning control to the caller. The other fork, which corresponds to the blank token that was introduced by the boundary transition, terminates control.

It should be noted that the topology of figure 3 does not enforce an ordering on the incoming packets. If two packets with a correct checksum arrive before the split transition fires, then the choice as to which packet is chosen as eligible is non-deterministic. The Internet Protocol, which is the (N-1)-service, is datagram oriented, and may deliver packets out of order. Figure 3 demonstrates very simply that the order of incoming packets is unimportant to the specification.

These two nets compose the entire (N-1)-interface specification. Conceptually, one could view the two transition boundaries presented in these nets as being joined between

(N-1)-interface: receive a packet



(1) ep: t2's *packet* is for t1's connection
mr: *t1.seg = t2.packet.data*

Figure 3. RCVPKT

two N-peers, as these are the only transition boundaries in the specification.

N-interface

Only a partial N-interface specification is made. In particular, of the several calls that processes may make upon TCP in the TCP specification, only one call, the *open*() call is specified, as this is the only call that actually deals with connection establishment. The form of the *open*() call described here is somewhat more limited than the specification presented in [TCP]. The specification herein permits a user process to issue an open request only for those connections which are in the CLOSED state.

The *open*() call takes the following input parameters:

- *port_type lport*, which indicates the local port that the user process wants to use as its end-point in a connection.
- *socket_type fsok*, which indicates the foreign socket that the user process wants to connect with. If this is omitted or partially given, then any foreign socket matching the requirements can establish a connection.
- *boolean active*, which indicates if an active or passive open should be done.
- *usr_sig_type timeout*, which indicates, if given, a user signal handling routine to be notified when a segment is not be acknowledged within a certain time limitation.
- *precedence_type prc*, which indicates the precedence level that the connection should have.
- *security_type sec*, which indicates the security/compartment level that the connection should have.
- *options_type options*, which specifies any TCP options to be used.

When a user process issues an *open*() call, an *open request* is issued on behalf of the user process. This results in the OPEN net (figure 5) being instantiated, given the parameters specified by the user process for the *open*() call. During the execution of OPEN, either an error code or a connection handle is returned to the user process. If a connection handle is returned, then an associated *tcb* has been instantiated for the user process.

N-protocol

The actions of the N-protocol for a particular connection start with figure 5, the OPEN net, as its main structured Petri net. Before describing OPEN, it is necessary to digress and introduce another net, MAIN, in figure 4.

If a segment arrives for a connection that does not exist, some method for rejecting the segment must exist. This generally requires some global knowledge of all of the connections that are known on the local host. Since RCVPKT receives all incoming packets, and only known connections instantiate RCVPKT to fetch packets, packets arriving for non-existent connections will "stack-up" at the place feeding the split transition in figure 3. Clearly, this is incorrect behavior. Hence, figure 3, although correct on a per-connection basis, and correct as a specification of part of the (N-1)-interface, is not actually invoked by the system. Rather, connections wishing to fetch the next packet meant for them instantiate the MAIN net at the RCVPKT entry place.

The MAIN net (figure 4) specifies several parts of the system. When the system starts, it instantiates MAIN at the TCP entry place. MAIN.TCP is viewed as a "single entry, zero exit" net. In short, an instantiation of MAIN.TCP never returns. The TCP entry place begins by establishing a new context containing a list of connections, which is

initially empty. It then proceeds to place where it can enable any of three split transitions. Once any of these transitions fire, control loops back.

If the OPEN entry place is instantiated in MAIN, a split transition fires which checks if the connection specified by the parameters ***lport*** and ***fsok*** is present in the connection list. If so, the *ISCONN* error is raised and this is returned to the caller. Otherwise, a connection handle is associated with a new connection for ***lport*** and ***fsok***. This connection is added to the list of known connections, and the connection handle is returned to the caller.

If the CLOSE entry place is instantiated in MAIN for a connection, a split transition fires which removes the connection from the list of known connections, and flushes any queues and releases any resources associated with the connection.

Independent of the instantiation of the OPEN and CLOSE entry places, if a packet arrives for a connection that is not present in the list of known connections, a third split transition fires, which invokes the BADSEG net to reject the segment.

The remainder of the MAIN is essentially the RCVPKT net. The RCVPKT entry place in MAIN corresponds to the single entry place in the RCVPKT net, and the split transition has the same enabling predicate and firing actions[1]. The interesting interaction, which was the motivation for merging the RCVPKT net into main, is that the place which supplies incoming packets to the RCVPKT control path, can also supply incoming packets to the BADSEG control path. Since this place feeds both transitions, either may make use of incoming packets, depending on how well their enabling predicates are satisfied.

Now that MAIN has been discussed, we can proceed with the parts of the specification dealing with individual connections. As discussed above, an ***open request*** results in OPEN being instantiated with the appropriate parameters. OPEN first checks the validity of the request.

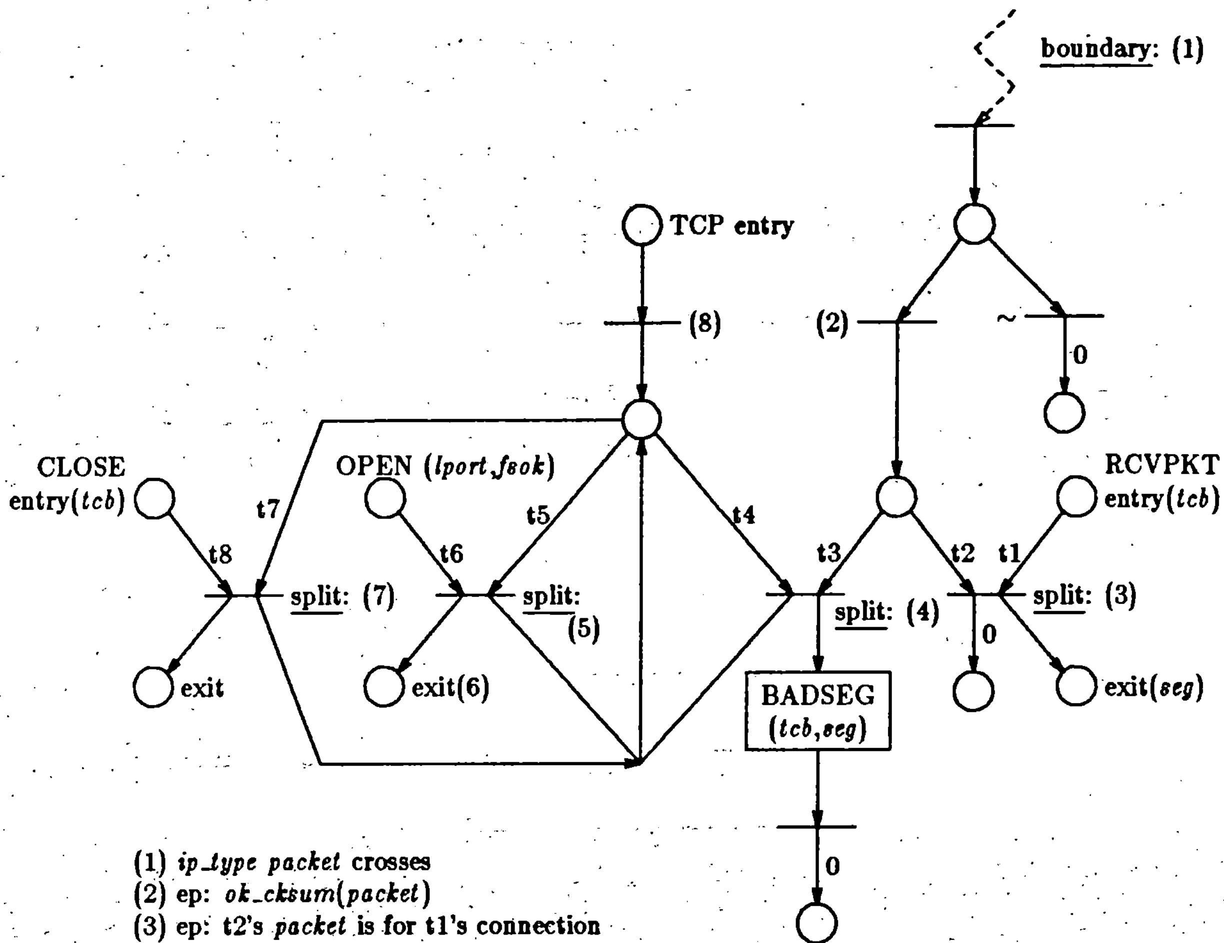
This check can be summarized as:

1. If the user process is not allowed to access ***lport***, or is not allowed to use the indicated ***prc*** and ***sec*** levels, then the error *DENIED* should be raised.
2. If ***active*** is set, but ***fsok*** is not fully specified, then the error *NOPEER* should be raised.
3. If there are insufficient resources to handle this request, then the error *NOBUFS* should be raised.
4. Otherwise, the request is considered valid.

If the request is not valid, the appropriate error is given to the user process and control returns as well. Otherwise, the MAIN net is instantiated at the OPEN entry

[1] The observant reader will notice a slight deficiency in figure 4. Due to unfortunate space limitations, a transition is missing immediately after the RCVPKT entry place. This transition creates the ***segment type seg*** which is returned. The author apologizes for this inconsistency.

N-protocol: connection oracle



(1) *ip_type packet* crosses
(2) ep: *ok_cksum(packet)*
(3) ep: t2's *packet* is for t1's connection
mr: *t1.seg = t2.packet.data*
(4) ep: t3's *packet* is not for any connection known in the context of t4
cr: new *tcb* initialized for dummy connection
(5) mr: if the connection specified by t6's *lport* and *fsok*
is known in the context of t5, then raise *ISCONN*
otherwise, associate a connection handle in the context of t6
and add the connection to the list in the context of t5
(6) return error or connection handle
(7) mr: remove t8's connection from t7's list of connections
flush any queues and release any resources associated with t8's connection
(8): cr: new list of connections, initially empty

Figure 4. MAIN

place. If MAIN.OPEN returned an error, that error is given to the user process and control returns. Otherwise, the connection handle returned by MAIN.OPEN is given to the user process. Next, the OPEN net checks to see if the user process wanted an active open. If so, a SYN segment is sent, and the connection enters the SYN_SENT state (by instantiating the SYN_SENT named net). If, instead, a passive open was requested, the connection enters the LISTEN state. Eventually, the path taken returns and the MAIN net is instantiated at the CLOSE entry place. When MAIN.CLOSE returns, control returns as the connection has now entered the CLOSED state and no longer exists.

N-protocol: open a connection

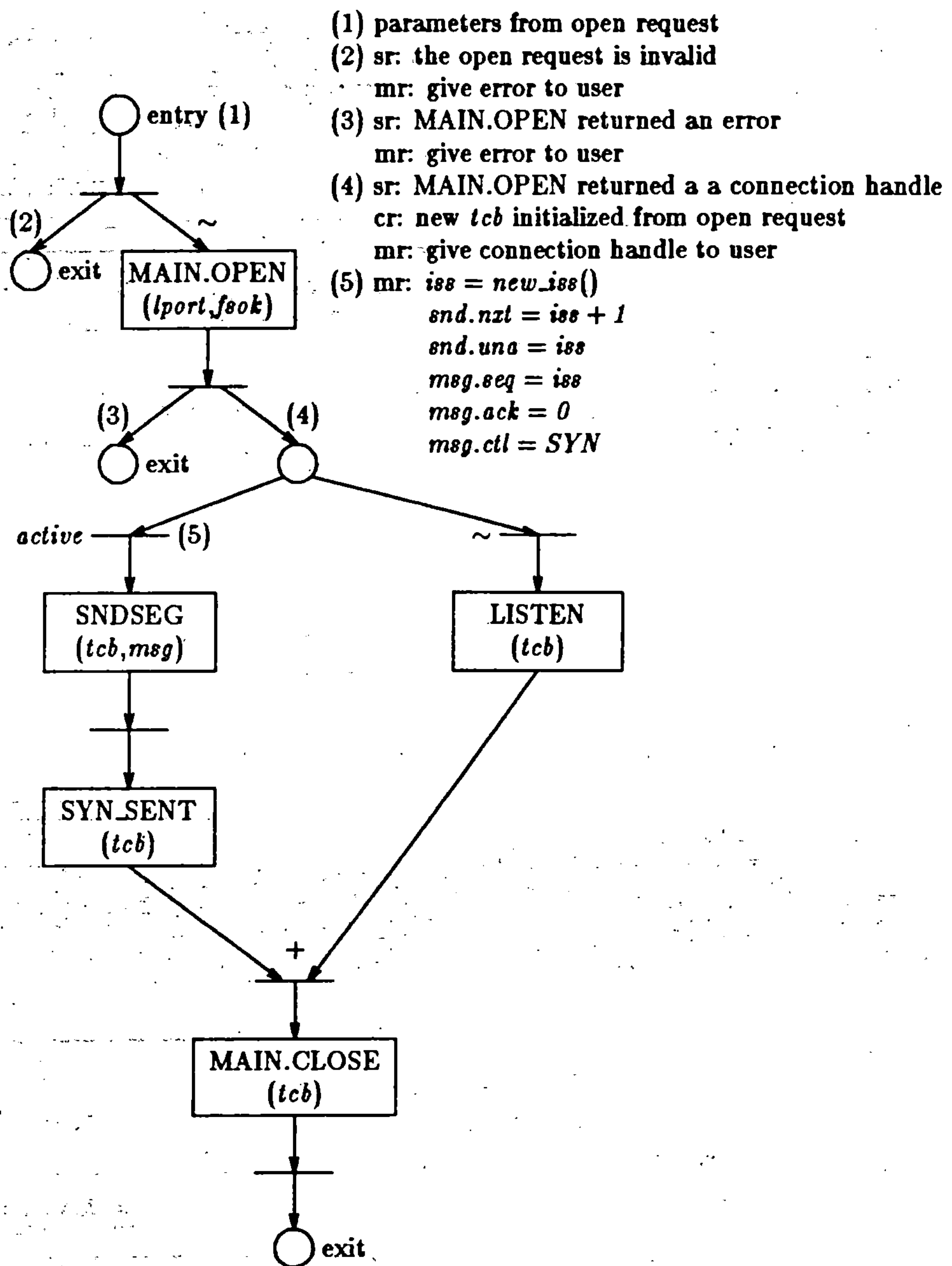


Figure 5. OPEN

The LISTEN named net (figure 6) is used to process a connection that is in the LISTEN state. First, RCVSEG is instantiated to await and return an incoming segment for this connection. A RST is explicitly checked for. If present, LISTEN ignores the segment. If an ACK is present, the segment is rejected. If a SYN is not present, the segment is ignored. If a SYN is present, but precedence and security/compartment considerations are not satisfied, then the segment is rejected, otherwise, the connection enters the SYN_RCVD state, and a SYN/ACK is sent as the second part of the three-way handshake. If more of TCP were being considered, other parts of the segment might be

N-protocol: LISTEN state

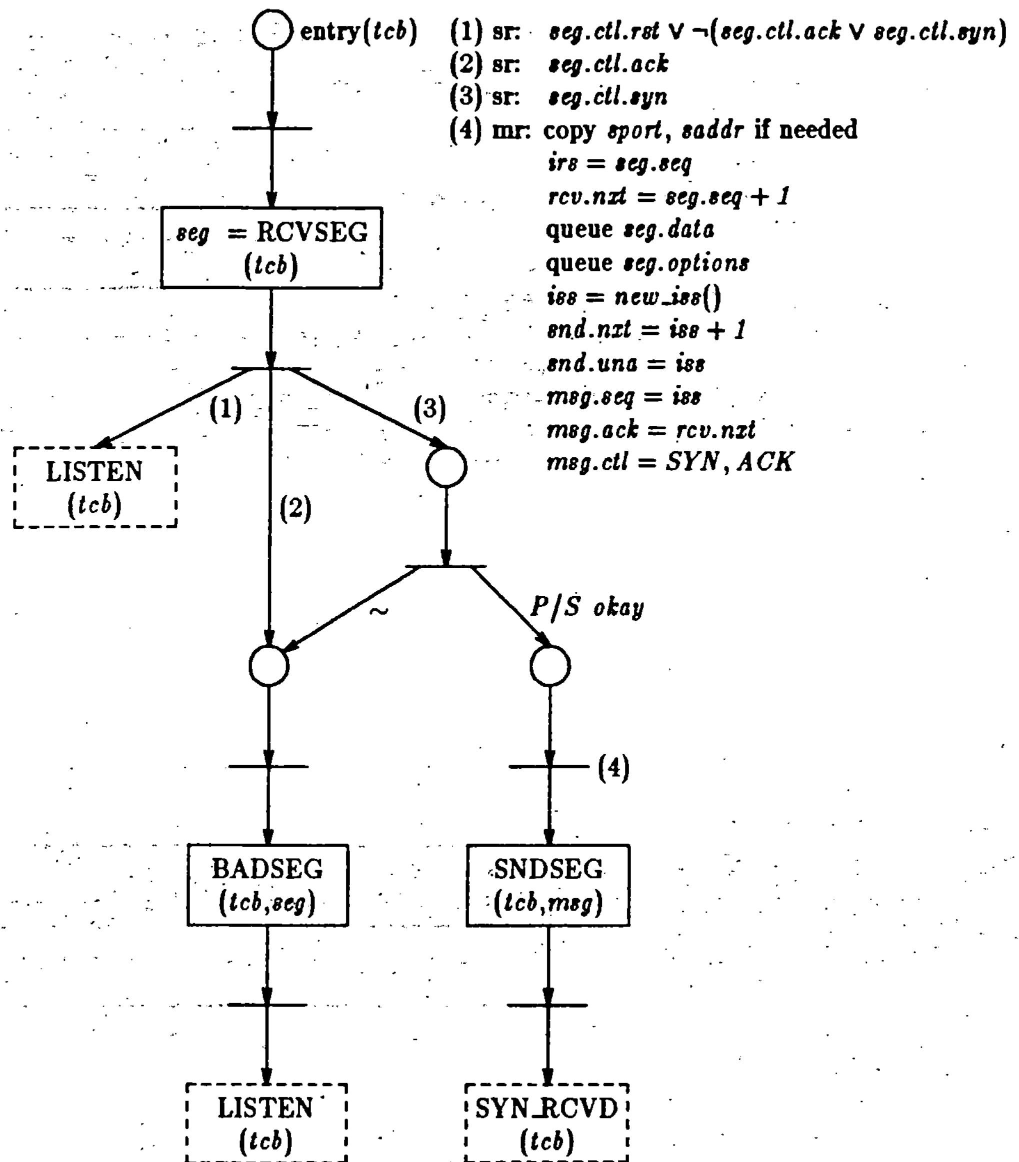


Figure 6. LISTEN

processed prior to entering the SYN_RCVD state.

The SYN_SENT named net (figure 7) is used to process a connection that is in the SYN_SENT state. First, RCVSEG is instantiated to await and return an incoming segment for this connection. If the segment acknowledges a segment not belonging to this instance of this connection, the segment is rejected. Otherwise the presence of a RST and a SYN is checked for. If a RST is present, and this is an acknowledgement, then the user process is informed that the connection was rejected, and control returns (to the OPEN net). Otherwise, the connection remains in the SYN_SENT state (the segment is ignored). Precedence and security/compartment considerations are then checked for, if not satisfied, the segment is rejected. If the considerations are satisfied, then a response

N-protocol: SYN_SENT state

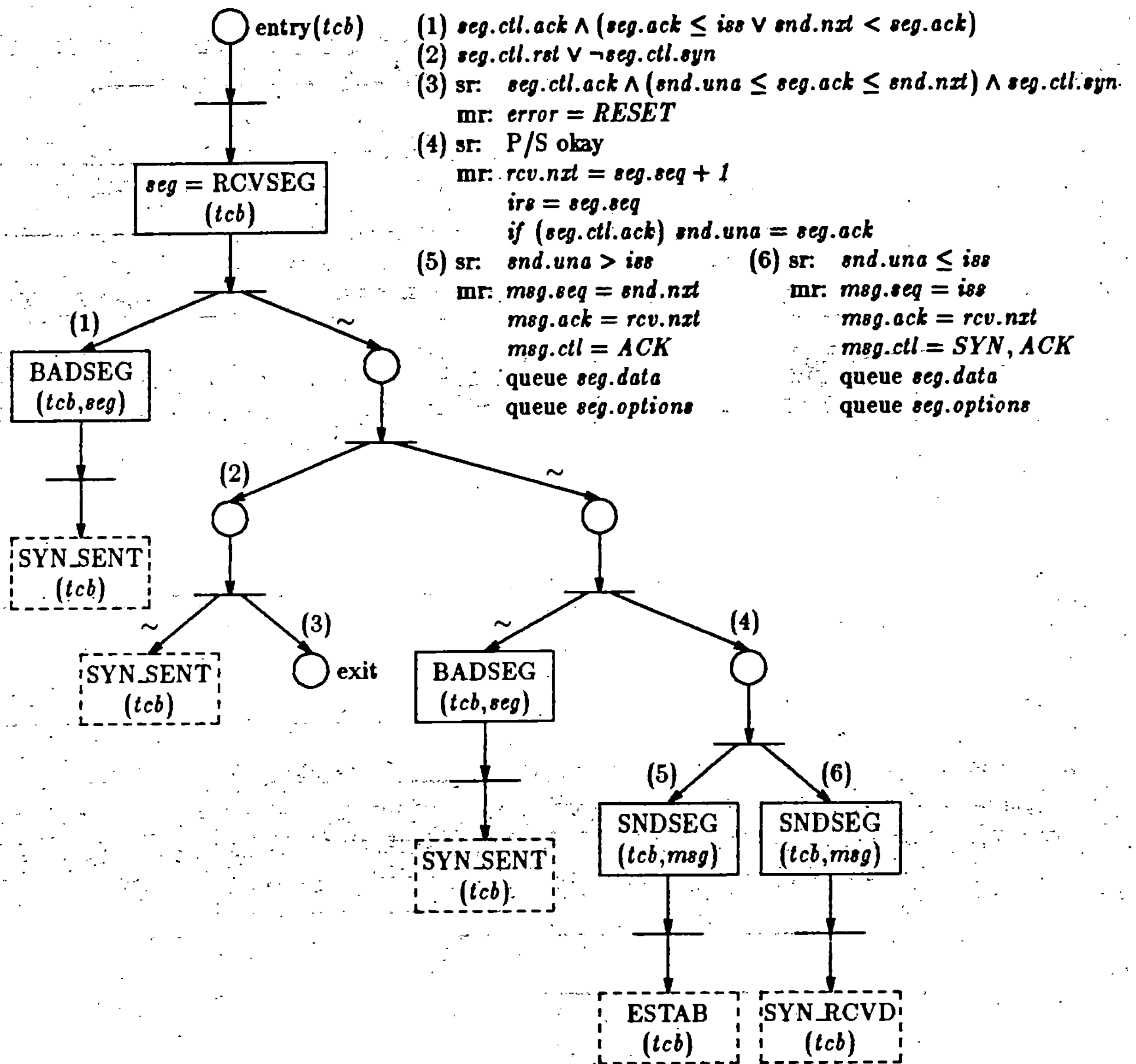


Figure 7. SYN_SENT

is sent, and the connection enters either the ESTAB or SYN_RCVD state, depending on whether our SYN has been acknowledged. Again, if more of TCP were being considered, other parts of the segment might be processed prior to entering the new state.

Figure 7 is rather sequential in nature. This need not be the case. A single, large switch-decision could be used to remove the sequential nature of the decisions which lead to the net's actions. This was not done for reasons of clarity. Depending upon the designer's interpretation of the trade-offs, the structured Petri net technique could be used to remove nearly all of the sequential nature of this invocation.

The SYN_RCVD named net (figure 8) is used to process a connection that is in the SYN_RCVD state. First, RCVSEG is instantiated to await and return an incoming

segment for this connection. If the segment is outside the window, the presence of a RST is checked for. If present, the segment is ignored; if not, a response is sent to force the foreign peer to re-transmit a valid segment. If the segment is inside the window, the presence of a RST is checked for. If present, and this connection was started with an active *open*(), the user process told that the connection has been refused, and control returns. Otherwise, the connection enters the LISTEN state. If a RST was not present, precedence and security/compartment considerations are checked, if not acceptable, the segment is rejected; otherwise the presence of a SYN is checked for. If present, the segment is rejected, the connection is closed, and the user process is informed that the connection has been reset; if not, the presence of an ACK is checked for. If not present, the segment is ignored. If an ACK is present, a check is made to make sure that the ACK is correctly acknowledging our SYN. If not, the segment is rejected. Otherwise the connection enters the ESTAB state. Again, if more of TCP were being considered, other parts of the segment might be processed prior to instantiating ESTAB.

At this point, four additional named nets should be presented: BADSEG, SNDSEG, RCVSEG, and TIMER. Due to space considerations, they are not presented in this paper. Rather, they are shortly described here. The BADSEG net rejects a segment by constructing a reply with RST set (but only if the segment being rejected did not have RST set).

The SNDSEG net is used to send a segment to the other N-peer. It instantiates SNDPKT to interact with the (N-1)-layer, and also adds the segment to a retransmission queue. The RCVSEG net is used to get the next segment from the other N-peer. It instantiates MAIN.RCVPKT to get the next packet for this connection, and removes from the retransmission queue any segments that are acknowledged by the incoming segment.

The TIMER net is responsible for handling retransmission. SNDSEG and RCVSEG call upon different entry places in TIMER to manipulate the retransmission queue for their connection. Further, TIMER makes use of a transition with a non-zero enabling time to model a time-out and issue repeated calls to SNDPKT to retransmit the first segment on the retransmission queue.

N-protocol primitives

The definitions of the predicates and routines used in the enabling predicates and firing actions of some of the transitions in the net are presented here.

The routine *do_chksum*() computes the TCP checksum for a segment and associated packet, and stores the checksum in the segment's *cksum* field. Similarly, the predicate *ok_chksum*() computes the TCP checksum for a segment and associated packet, and compares it to the value of the segment's *cksum* field. If the two values do not match, the routine returns *FALSE* otherwise it returns *TRUE*.

The *new_iss*() routine calculates an initial sequence number for a *tcb*.

The predicate *P/S okay* refers to the precedence and security/compartment conditions being satisfied.

N-protocol: SYN_RCVD state

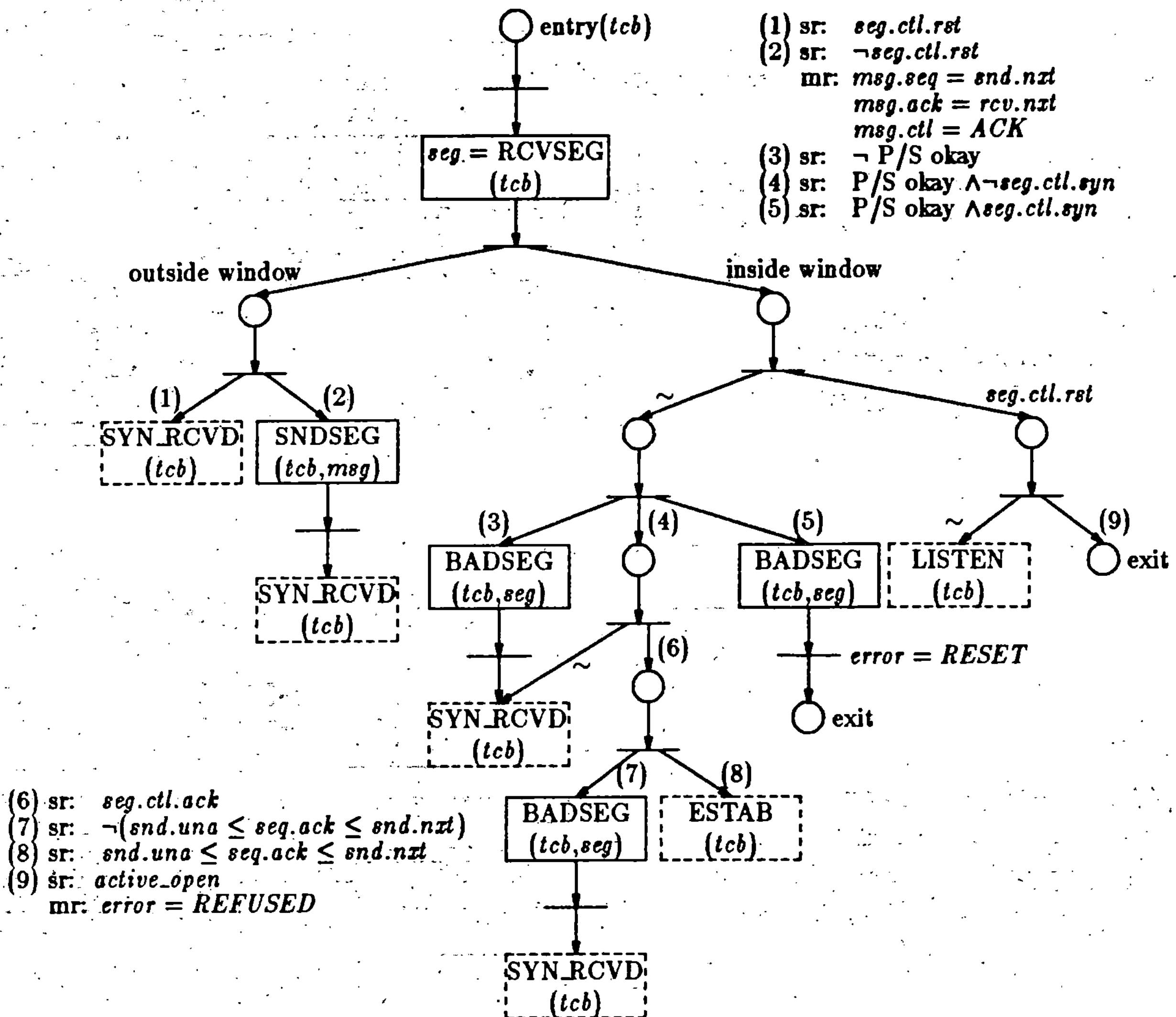


Figure 8. SYN_RCVD

Finally, the predicates ***inside window*** and ***outside window*** check to see if a segment is inside the valid window for a connection, checking the segment's sequence number and the *s_wnd_type* structure of the *tcb*.

6. Evaluation

This section makes an abbreviated examination of the preceding specification. In comparison to the full specification in [TCP], several observations can be made, which point to the advantages and weaknesses of the structured Petri net approach. In many ways, the structured Petri net approach is less ambiguous that its natural language counterpart. The flow of control for a particular state is more clearly defined. Despite the use of "structured" (i.e. rigorously indented) paragraphs in [TCP], ambiguities do arise.

These are in-escapable[1]. Although "structured", little hierarchy is present in [TCP], which results in rather repetitive groups of statements throughout the specification. In contrast, the structured Petri net specification does not suffer from these problems, as Petri nets are used to convey the bulk of the meaning. Even so, the structured Petri specification does make use of several predicates, functions, and procedures which are presented in natural language. It is emphasized that the sPn technique does not seek to eliminate the use of natural language as a part of the specification, but rather to introduce a rigorous approach which is more capable of capturing the spirit of the protocol and is less ambiguous than less formal methods.

A difficult problem in presenting a specification is "knowing when to stop". That is, at what point has the specification given the full functional description, and further discussion on the specification's part is actually constraining possible implementations? Both specifications do rather well in this regard, although [TCP] tends to do somewhat better. In providing a less ambiguous specification, the structured Petri net approach has taken the liberty to make several things more concrete, so as to avoid possible mis-interpretation. It is not clear if this has crossed the line from functional specification to implementational constraint.

Finally, a criticism made of many specification and verification efforts is the lack of ability to properly describe the behavior of the protocol when more than one connection is active. For the specification presented in this paper, the use of colors to represent connections, and the clean semantics of colors in structured Petri nets, permits a natural description of the protocol's activities. For verification purposes, if one could demonstrate that initial connection handling for a single connection was handled correctly, and one could demonstrate that the interaction between connections (i.e. colors) did not disturb this property, then one can prove that multiple connections are handled properly by the specification.

Conclusions

The three-way handshake, which is used by the Transmission Control Protocol as a means of initial connection establishment, has been specified using structured Petri nets. The specification has demonstrated the ability of this technique to represent a significant portion of a reasonably complex protocol.

The specification is not without its faults however, and these faults demonstrate the weaknesses of the structured Petri net approach. This approach has apparently opted to sacrifice ease of verification for power of expression. Despite their relatively clean semantics and nice conceptual basis, structured Petri nets apparently do not lend themselves well to existing verification techniques.

[1] This section is not meant to be a critique of [TCP]. *Any* specification made using a natural language approach will suffer these problems.

Fortunately, as a specification tool, structured Petri nets have other uses. In essence, structured Petri nets are able to concisely present a protocol using the natural concurrency logic found in Petri nets, and the compact data representation and manipulation capabilities found in programming languages. This means that the spirit of a protocol can be easily represented using the structured Petri net approach without undue clutter.

Further research in the area of the structured Petri net approach must clearly concentrate on its greatest weakness: the lack of verification concepts. Fortunately, the verification of specifications using Petri nets is well understood (e.g., [BERT82]). Further, we note that structured Petri nets bear a close resemblance to the "presentational model" in [KELL76]. If structured Petri nets can be shown to have convenient or powerful verification techniques applicable, then their use in protocol specification would be much more desirable.

REFERENCES

[BERT82] G. Berthelot, R. Terrat. Petri Net Theory for the Correctness of protocols. *IEEE Transactions on Communications 12*, COM-30 (December, 1982), 2476-2505.

[IP] Internet Protocol. Request for Comments 791. Appearing in Internet Protocol Transition Workbook, Network Information Center, SRI International, 1981.

[KELL76] R.M. Keller. Formal Verfication of Parallel Programs. *Communications of the ACM 7*, 19 (July, 1976), 371-384.

[MROSE83B] M. Rose. An Introduction to Structured Petri Nets. *Technical Report 218* (October, 1983), Department of Information and Computer Science, University of California, Irvine.

[PETE77] J.L. Peterson. Petri Nets. *Computing Surveys 3*, 9 (September, 1977), 224-252.

[SUNS78B] C.A. Sunshine, Y.K. Dalal. Connection Management in Transport Protocols. *Computer Networks 16*, 2 (November, 1978), 454-473.

[SYMO80A] F.J.W. Symons. Introduction to Numerical Petri Nets, a General Graphical Model of Concurrent Processing Systems. Appearing in Communication Protocol Modeling, C.A. Sunshine, editor, Artech House, 1981.

[TCP] Transmission Control Protocol. Request for Comments 793. Appearing in Internet Protocol Transition Workbook, Network Information Center, SRI International, 1981.