

On Defining the Service Provided by TCP

Jonathan Billington

Bing Han

Computer Systems Engineering Centre
University of South Australia,
Mawson Lakes, SA 5095, Australia
Email: Jonathan.Billington@unisa.edu.au
Bing.Han@postgrads.unisa.edu.au

Abstract

The Transmission Control Protocol (TCP) is an important transport layer protocol providing a reliable data transfer service to support many applications running over the Internet such as the World Wide Web. It is therefore important to define the intent of TCP. This is achieved by describing the services it is intended to provide in an abstract way, known as a *service definition*. Several attempts have been made to define the services provided by TCP, however, they are all inadequate in various ways. In particular, most of them follow the applications interface defined in the TCP Request for Comments (RFC 793) too closely, making them rather implementation specific. The aim of our work is to provide an appropriately abstract definition of the TCP service, following the guidance of the Open Systems Interconnection service conventions, and to make it as general and as complete as possible. We define the service in terms of a set of 15 *service primitives* and their sequences, for TCP's main features of connection establishment, data transfer, urgent data transfer, orderly connection release, and aborting connections. The choice of primitives is discussed in terms of their relationship with the Interface Definition of RFC 793. We formalise the TCP service using Coloured Petri Nets, and from it generate the global sequences of service primitives for TCP's Connection Management services (including aborts). We believe this is the first time this has been achieved.

Keywords: TCP, Service Definition, Finite State Automata, Coloured Petri Nets.

1 Introduction

Many applications in the Internet such as the World Wide Web, E-mail and remote login (Telnet) run over the Internet's reliable transport protocol known as the Transmission Control Protocol (TCP). The operation of TCP was specified 20 years ago in a so called Request For Comments document (RFC 793) (Postel 1981), and then improved and modified (Braden 1989, Jacobson, Braden & Borman 1992, Paxson 1999, Allman, Paxson & Stevens 1999, Floyd & Henderson 1999). The Internet protocol specifications are now developed and managed by the Internet Engineering Task Force (IETF 2002). The history and philosophy of developing Internet protocols has been implementation oriented. Protocol descriptions are provided using narrative descriptions, possibly supported by the use of state transition diagrams or tables. IETF protocol documents may also include

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

descriptions of interfaces. For example, RFC793 provides a narrative description of TCP (supported by an incomplete state transition diagram) and its interface to applications. Both descriptions are stated to be from an implementation point of view. TCP is a relatively complex protocol, and the number of bugs reported in implementations (Paxson 1999) in March 1999 spans 60 pages.

This (and other) experience leads us to believe that a more formal and rigorous approach to protocol design may prove beneficial. In 1981, when the first TCP RFC was issued, there was very little tool support for formally specifying and analysing complex protocols. Twenty years later, there is significant tool support, and the capacity to fully specify complex protocols using formal techniques. However, formal verification of these protocols is still a significant challenge.

Verification of a protocol requires formal specifications of three elements: the service provided by the protocol to its users (known as a *service specification*); the protocol being considered (the *protocol specification*); and the characteristics of the medium over which the protocol is going to operate (the *underlying service specification*). The verification step compares the service provided by the protocol operating over the underlying service, with its service specification. Thus a first step towards verifying TCP is to provide a formal specification of the service TCP provides to its applications.

Unlike protocols specified by the International Organisation for Standardisation for Open Systems Interconnection (OSI), RFC 793 does not include (or refer to) a service definition. Instead it provides a user interface definition comprising a set of six calls (Open, Send, Receive, Close, Status, Abort) that the user could make to a TCP entity. A description of what information is transferred from the TCP entity to the user is less well defined. The user interface calls include implementation details that are not important in a service specification, which needs only to relate to actions that have end-to-end significance. For example, the TCP/User interface Open call includes a Passive Open, which has only local significance. It requests that a TCP server entity be established in the local host, ready to receive connection requests from remote hosts. Thus it does not convey end-to-end information. The Receive and Status calls also have only local significance.

There have been previous attempts to write service definitions for TCP, dating back to 1986 (Cass & Rose 1986) and revised in (Rose & Cass 1987) and (Pouffary & Young 1997). In his book on Computer Networks, Tanenbaum (Tanenbaum 1996) also discusses a transport service that is considered appropriate for TCP. However, we believe that these attempts are too close to interface definitions as they

also include purely local interactions. For example, LISTEN and RECEIVE service primitives are defined in (Tanenbaum 1996) (with similar primitives defined in (Pouffary & Young 1997)), which map well to the TCP interface calls but only have local significance. They are thus not needed in a service specification.

Work on formalising the TCP service appears in (Murphy & Shankar 1991) and (Smith 1996). The first paper specifies the connection management service for a transport layer, in the spirit of TCP connection management, and formalises it using a state transition approach. However, the graceful release of connections is not included, and again events which may only be considered local are included. It also includes events (such as detection of old connection attempts) that would normally be considered to be internal to the layer, and hence not reported to the user. (Smith 1996) gives a service specification for TCP using an automaton model. However, the specification is incomplete as it only presents some of the steps for the specification. Also, the primitive events for simultaneous open are missing and there is no provision for user abort or urgent data.

In this paper, we define the TCP service according to the established conventions for defining OSI services (ITU-T 1993). The definition is general in that it includes all TCP services: connection establishment, normal data transfer, urgent data transfer, graceful connection release and user and provider abort. The conventions introduce the idea of defining a set of abstract events known as *service primitives* for encapsulating communication between the user of the service (the *service-user*) and the provider of the service (the *service-provider*).

We follow the OSI service conventions by only including primitive events that have end to end significance. We define a set of 15 service primitives and specify their sequences at the local interfaces (client and server) using state tables.

The TCP service is formalised with Coloured Petri Nets (Jensen 1997). This allows us to define the end to end relationships between service primitives (i.e. the *global sequences*) using reachability analysis and automata reduction. We demonstrate this for the connection establishment, graceful release and abort services.

This paper is organised as follows. Section 2 defines the TCP service in terms of its service primitives and local sequences. Section 3 introduces Coloured Petri Nets (CPNs) while section 4 presents the formal model of the TCP service. We generate the global sequences of service primitives in section 5 for the connection management services, and present it as an automaton. Section 6 concludes this paper and discusses future work.

2 TCP Service Definition

In defining the TCP service we follow the OSI Transport Service Definition (ITU-T 1995b), except that (ITU-T 1995b) does not include an orderly release service. We therefore also make use of the OSI Session Service Definition (ITU-T 1995a) for the orderly connection release service.

According to OSI-service conventions (ITU-T 1993), a service definition describes the behaviour of a service provider observed by its service users. It does not prejudice the internal behaviour of the service provider. A service definition comprises two main parts: a set of service primitives and their sequencing constraints at the service/provider interface. In this section, we define the two parts of the service that TCP and its underlying protocols (service provider)

offer to application protocols (service user). However, firstly we provide some justification of why our approach is valid for TCP, which was defined in a very different environment from that of the International Standards work.

2.1 Rationale for choosing TCP Service Primitives

The OSI Service Conventions adopt an approach to service definition that is predicated upon making the definition as implementation independent as possible. This is desirable, so that implementers can innovate, and take advantage of characteristics of their local operating systems. The early work on the Internet took a complementary approach, providing as much guidance as possible on the implementation of both the protocol and the application interface. Both approaches are valid in their own contexts. For OSI protocols, the interface definitions are not provided, as it was not considered appropriate to standardise them. In the work of the IETF, service definitions are missing.

Service definitions are important for a number of reasons (Vissers & Logrippo 1986, Smith 1996) including their ability to define user requirements at an abstract level and their necessity for protocol verification. The OSI conventions strongly make the case for service primitives to be defined which involve only end-to-end behaviour. Any activity across the interface that is only a local matter is part of the interface definition, and has no place in a service definition. It therefore behoves us to take this into account when defining a service.

Perhaps the most contentious issue when defining the service arises out of the interface definition provided in RFC 793 (Postel 1981) for establishing connections. RFC 793 provides for both an *active open* and a *passive open* interface call from the application. The active open is used by a client to initiate a connection request to a server. This is very easily mapped to a Connect request primitive. However, the passive open is for a server to indicate its willingness to accept connections from remote hosts, *but without informing those hosts over the network*. It is thus a local matter. We therefore do not have a primitive corresponding to the passive open call of the TCP interface. We also differ from the OSI session service, in that we do not provide for the TCP server user to refuse a connection request, as part of the connect response primitive. This is because the only mechanism that TCP provides in its interface definition, is for the connection to be established if a passive open has occurred, or for it to be aborted at the TCP level (i.e., provider abort) if a passive open has not occurred or if it has been followed by a *close* call.

2.2 TCP Service Primitives

The Application/TCP interface is where the TCP service is provided and accessed, by invoking service primitives. In this subsection, 15 service primitives and their parameters are defined for each component service of TCP, namely connection establishment, data transfer (including urgent data transfer), connection release and abort.

Table 1 summarises the primitives, their parameters and significance. The first column shows the type of the service. The second column lists each primitive, which is named according to the established conventions for naming OSI service primitives (ITU-T 1993). The name comprises three components: an initial or initials, indicating the layer that provides the service

Service	Primitive	Parameters	Significance
Connection establishment	TCP-CONNECT.req	local socket, foreign socket, quality of service, user data	A connection open request is submitted.
	TCP-CONNECT.ind		A connection open indication is delivered.
	TCP-CONNECT.res		A connection open response is submitted.
	TCP-CONNECT.cnf		A connection open confirm is delivered.
Normal data transfer	TCP-DATA.req TCP-DATA.ind	user data	A data transfer request is submitted. A data transfer indication is delivered.
Urgent data transfer	TCP-URGENT_DATA.req TCP-URGENT_DATA.ind	user data	An urgent data transfer request is submitted. An urgent data transfer indication is delivered.
Orderly release	TCP-RELEASE.req	user data	A connection release request is submitted.
	TCP-RELEASE.ind		A connection release indication is delivered.
	TCP-RELEASE.res		A connection release response is submitted.
	TCP-RELEASE.cnf		A connection release confirm is delivered.
User abort	TCP-U_ABORT.req TCP-U_ABORT.ind	reason, user data	An abort request is submitted by a user. An abort indication invoked by a user is delivered.
Provider abort	TCP-P_ABORT.ind	reason	A connection abort indication invoked by the service provider is delivered.

Table 1: TCP service primitives

(in our case we will use TCP); a *service type*, (e.g., CONNECT or DATA); and a *primitive type*, i.e., request, indication, response and confirm (abbreviated to req, ind, res, and cnf). Primitives submitted by the service user are request and response type primitives. Primitives delivered by the service provider are indication and confirm type primitives.

In Table 1, the connection establishment service involves invoking four CONNECT primitives: TCP-CONNECT request, TCP-CONNECT indication, TCP-CONNECT response, and TCP-CONNECT confirm. It is a confirmed service, that is, it requires an explicit confirmation. The data transfer service (including normal and urgent data transfer) is an unconfirmed service as it has only request and indication type primitives. The orderly release service is a confirmed service with four RELEASE primitives. The abort service can be initiated either by the service user or provider, and it is an unconfirmed service. User abort requires request and indication type primitives, whereas provider abort has only an indication.

2.3 Introducing Primitive Sequences

An important part of the service specification is the definition of the possible sequences of primitives that can occur at the user/provider boundary. Primitives are invoked at each service access point (SAP) of TCP. SAPs are identified by an address which is either the client socket or the server socket.

Figure 1 shows the normal scenario for a client establishing and releasing a connection. On the left side are the primitives that occur at the client SAP (designated SAPc). They are ordered in time down the page, occurring in the sequence: 1, 4, 5, 7 and 10. This is known as a *local sequence*, as it is local to one SAP. Similarly the right side of the figure depicts the sequence of primitives for the server SAP (SAPs). A global view of the sequences takes into account the timing relationships of primitives at different SAPs. As shown in Figure 1, there is a sequence of 10 primitives, in the order 1-10, involving the exchange of four TCP-CONNECT primitives followed by the exchange of two TCP-DATA primitives and then the exchange of four TCP-RELEASE primitives. The figure just shows one possible *global sequence*. There are, of course, an infinite number of sequences, as there is no limit to the amount of data (and urgent data) that can be transferred in either direction. Note that the RELEASE requests can be initiated by either side and that the sequence depicted in Figure

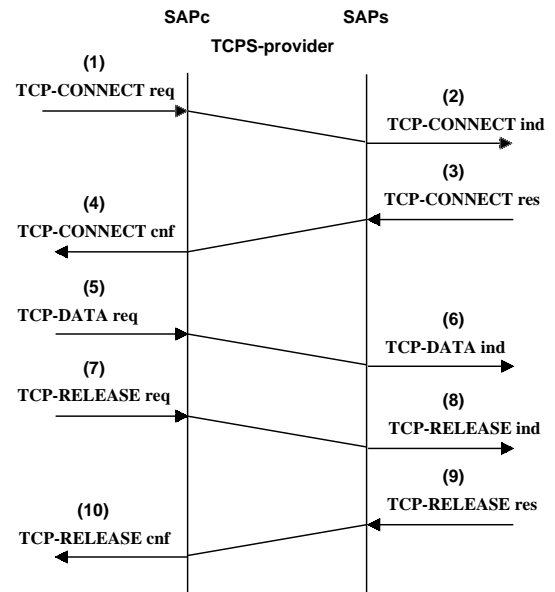


Figure 1: Time Sequence Diagram for the normal sequence of primitives

1 has not included the Abort primitives, which can occur at anytime after a connection request has occurred at the client SAP, or a connection indication has occurred at the server SAP.

Time sequence diagrams provide a very convenient and straight forward way of illustrating sequences of primitives when the sequence is small. However, they are inconvenient when the sequence is long, and inadequate for specifying infinite sequences and unbounded numbers of sequences.

We use a two step process to define the complete set of (global) sequences. As is usual in OSI service definitions, we firstly specify the local sequences at each service access point using a finite state automaton (FSA), represented by a state table. Once the local sequences are defined for each SAP, the global sequences can be obtained by linking together the FSAs for the client and server SAPs.

We shall shortly define the legitimate primitive sequences that can occur at the local client and server SAPs, but firstly we will discuss some of the decisions that have been made in defining them as we have.

2.4 Considerations for defining Primitive Sequences

As previously discussed, no primitive is provided for the Passive Open Call (defined in RFC 793) used by TCP servers to express the server's willingness to accept connection requests. This is considered a local matter, because it does not initiate communication with a peer, nor affect other end-to-end communication. In the case where the server is not ready to accept a call, the TCP server entity will be in the Closed state, and the TCP SYN message (used for opening connections) received from the client will result in a reset message being returned by the server. This will be mapped to a provider abort (TCP-P_ABORT indication) primitive at the client, with no primitives occurring at the server side. If the server side is willing to receive connections, then its TCP entity will be in the Listen state, and will return a SYN-ACK in response to a SYN initiated by the client application's TCP-CONNECT request. The receipt of the SYN at the server, allows foreign socket information to be indicated to the server application, resulting in a TCP-CONNECT indication. The TCP-CONNECT response can be mapped to the SYN-ACK, which when received by the client, indicates successful connection establishment at the client, and hence the occurrence of a TCP-CONNECT confirm.

We didn't consider that orderly release could occur until after the connection is established. This is because RFC 793 specifies that on receipt of a close call from the application when in the SYN_SENT state (the state TCP enters after sending out a SYN), all state variables are deleted, 'error: closing' responses are returned to the application for any outstanding data in TCP's buffers, and the state becomes Closed. This is the same as aborting the connection. Thus in this case, both close and abort calls are interpreted as implementing the TCP-U_ABORT request primitive. Also according to RFC 793, when the TCP entity at the server side receives the SYN segment (TCP-CONNECT indication occurs) it sends out a SYN-ACK (TCP-CONNECT response occurs) and enters the SYN_RCVD state. If a server application closes the connection when TCP is in SYN_RCVD, it does so gracefully, after entering the Established state (if there is data to send). This indicates that a TCP-RELEASE request can only follow a TCP-CONNECT response. Therefore, there is no submission of a TCP-RELEASE request after the delivery of a TCP-CONNECT indication.

As is usual practice, we allow the orderly release service to be invoked immediately after the connection is established, without any data primitives occurring. This is reasonable, as it is possible that all data that the user wishes to transfer can be included with the TCP-CONNECT request, and the server may or may not want to send data.

If both users initiate orderly release simultaneously, each side invokes a TCP-RELEASE request followed by a TCP-RELEASE confirm, which then completes the sequence.

TCP Abort service can be invoked at any time after the connection is requested. At the client side, after the TCP-CONNECT request is submitted, TCP-U_ABORT and TCP-P_ABORT primitives can be invoked. At the server side, after the TCP-CONNECT indication is delivered, TCP-U_ABORT and TCP-P_ABORT primitives can be invoked.

Note that simultaneous open by both users is not considered in this paper, but is allowed by TCP.

2.5 State Tables for Local Primitive Sequences

We specify the legitimate sequences observed at each SAP by a finite state automaton (FSA). We define six states (see Table 2). IDLE is a SAP state in which a sequence begins and ends. OCP is defined for the client SAP, while ICP is defined for the server SAP. They indicate that the client or server is in the process of establishing a connection. State DTR indicates that the connection is established (according to the local SAP) and the local user can either send or receive data. States NDS and NDR are required for the orderly release service. State NDS indicates that the local user finishes sending data but can still receive data from the remote user. State NDR indicates that the local user will not receive data (the remote user has no more data to send), but can still send data.

State	Significance
IDLE	The connection does not exist.
OCP	Outgoing connection pending.
ICP	Incoming connection pending.
DTR	Data transfer ready.
NDS	No more data to be sent.
NDR	No more data to be received.

Table 2: The significance of the SAP states

The local primitive sequences are defined in two state tables (see Tables 3 and 4). In each table, the first two rows list the TCP service primitives and the first column lists the state in which a SAP can be. Each table entry corresponds to the resulting state after a primitive has occurred. For instance, if the client SAP is in state IDLE and a TCP-CONNECT request occurs, it changes state to OCP. If an entry is empty, it means that a state transition can not occur.

Besides defining the normal local sequences (see Figure 1), Tables 3 and 4 also incorporate all other local sequences under the assumptions outlined previously.

3 Coloured Petri Nets

The previous section has defined the sequences of primitives at each of the local SAPs. In order to define the global sequences we need to compose the automata. This is not such a straightforward task when taking urgent data and aborts into account. To do this we follow previous work on defining the OSI Transport Service (Billington 1983). In that work, the automata are connected by two queues, one for each direction of information flow between the client and the server.

In this paper the queues preserve the sequence of normal data entered into them, but allow urgent data to overtake any amount of normal data in the queue. The urgent data stream is also maintained in sequence. Orderly releases can be entered into a queue and maintain their position at the end of the queue. On the other hand, when aborts are entered into the queue, any amount of the data in the queue may be discarded.

The original work only applied to the OSI Transport Service, and hence did not include the orderly release service. It also used a semi-formal technique, known as Numerical Petri Nets (Symons 1978). Here we shall use the mathematically precise Coloured Petri Nets (CPNs) (Jensen 1997) to formally specify the TCP service, at a level of abstraction where

	TCP-CONNECT		TCP-DATA TCP-U_DATA		TCP-RELEASE				TCP-U_ABORT TCP-P_ABORT	
	REQ	CNF	REQ	IND	REQ	IND	RES	CNF	REQ/IND	
IDLE	OCP									
OCP		DTR								IDLE
DTR			DTR	DTR	NDS	NDR				IDLE
NDS				NDS					IDLE	IDLE
NDR			NDR					IDLE		IDLE

Table 3: Client SAP state table

	TCP-CONNECT		TCP-DATA TCP-U_DATA		TCP-RELEASE				TCP-U_ABORT TCP-P_ABORT	
	ind	res	req	ind	req	ind	res	cnf	req/ind	
IDLE	ICP									
ICP		DTR								IDLE
DTR			DTR	DTR	NDS	NDR				IDLE
NDS				NDS				IDLE		IDLE
NDR			NDR				IDLE			IDLE

Table 4: Server SAP state table

we only take the service primitive names into account, and not their parameters. This is sufficient for defining the global sequences of primitives.

CPNs are Petri nets enhanced with data types. Instead of black dots, Petri net tokens are allowed to be arbitrarily complex values. The net is annotated with inscriptions which relate to typing and moving the tokens around the net. CPNs also provide some features for supporting hierarchical net construction, which eases maintenance of the net descriptions.

In brief, a CPN may be considered to be an annotated directed bipartite graph comprising nodes of kinds *place* and *transition* and directed edges known as *arcs*. A place is typed by a *colour set* and contains collections (multisets) of data items called *tokens* of the same type as the place. A transition represents an event and may have a *guard* associated with it. The guard is a boolean expression. Arcs connect places to transitions and transitions to places, and are annotated by expressions comprising variables, constants and function images (terms built from a signature). For a transition, its *input places* are those having arcs going to the transition, and vice versa for the *output places*. A Transition is *enabled* when there are enough tokens on each input place to match each respective input arc expression when evaluated for a particular binding of its variables, and the transition guard evaluates to true for the same binding. When an enabled transition *occurs*, the tokens in its input places that are specified by the respective arc expressions are removed and the tokens specified by the expressions on the output arcs for the same binding are deposited into its output places. The distribution of tokens on the places of the CPN is known as a *Marking*. Further details are given in (Jensen 1997).

Design/CPN (University of Aarhus 1996) is a software tool that facilitates constructing and analysing a CPN model. It comprises a graphical editor, syntax checker, simulator and reachability analyser. Design/CPN uses a variant of Standard ML (Ullman 1994) to provide net inscriptions and declarations. It also supports hierarchical constructs. We use it to construct our model of the TCP service as demonstrated in the next section.

4 Modelling the TCP Service

Our main purpose for modelling the TCP service using CPNs is to generate the global sequences of service primitives. We therefore model the TCP service at a high level of abstraction, where we do not model the parameters of the primitives. The CPN model of the TCP service consists of five pages, as shown in Figures 4 – 8. They are organised in a hierarchy as shown in Figure 2 with TCPS_Overview as the top page. TCPS_Overview utilises four subpages that specify the behaviour for the Connection Establishment, Data Transfer, Orderly Release and Abort services.

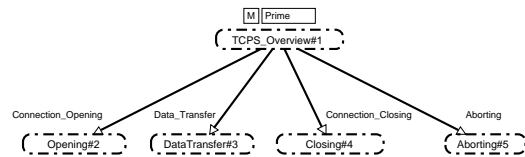


Figure 2: Hierarchy page

4.1 Declarations

The declarations for the CPN pages described in the next subsections are shown in Figure 3. Three types (colour sets) are defined: *States* comprising the six SAP states plus the state CLOSED; *Message* which defines the five messages that can be entered into the queues between the client and server automata; and *Meslist* which stores the messages in the queues in order of arrival. State CLOSED is used to indicate the end of a sequence while state IDLE indicates the beginning of a sequence. This is because we only need to consider the sequences for a single instance of a connection. Three variables are defined which run over these types: *s* for states; *mes* for messages; and *q* to represent a list of messages. The two functions *rm* and *member* are used in Figure 6 to model urgent data overtaking normal data. Function *rm* takes a message (*mes*) and a queue (*q*) as its argument. It removes the first appearance of message, *mes*, from the list. Function *member* takes a message and a list as its argument. It recursively searches the list for

```

color State=with IDLE | OCP | ICP | DTR | NDS | NDR |
              CLOSED;
var s: State;
color Message=with CONNECT | DATA | UDATA |
                RELEASE | ABORT;
var mes: Message;
color Meslist=list Message;
var q: Meslist;
fun rm(mes,q)=
  let val p=fn h => h=mes
      fun addhd(p,[])=[]
        | addhd(p,h::t)=if p h then t else h::addhd(p,t)
  in addhd(p,q)
  end;
fun member(mes,[])=false
  | member(mes,h::t)=mes=h or else member(mes,t);

```

Figure 3: Declarations

mes. It returns true if the message is in the list or false if not (or if the list is empty).

4.2 The Top Page

In Figure 4, the two places *SAPc* and *SAPs*, typed by colour set *State*, model the state of the SAP for the client and the server. The initial marking of each place is *IDLE*. The four transitions have “HS” tags and are substitution transitions, each modelling a type of TCP service, as indicated by the name of the transition. Each substitution transition is linked with a lower level page (subpage) and can be considered as a macro expansion. For example, transition *Connection_Opening* in Figure 4 is linked with subpage *Opening* (see Figure 5) via the so called *Port Places*, *SAPc* and *SAPs*.

Each of the subpages has a similar structure. They all comprise 4 places. Two of the places are the two port places encountered in the top page. The two extra places, *C_S* and *S_C*, model the TCP service provider as a pair of queues linking the client and server FSAs. *C_S* indicates that the transmission direction is from the client to the server and vice versa for *S_C*. They are fusion places (indicated by *FG* in a box near the place), which means that they are the same as those that appear on other pages with the same name. These queuing places are typed by *Meslist* with their initial marking being an empty list.

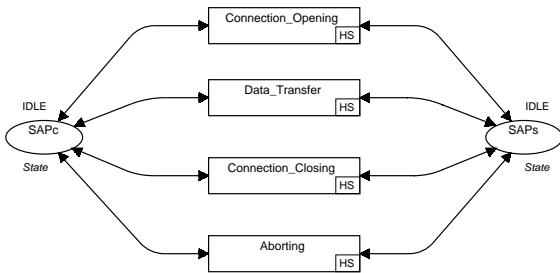


Figure 4: TCP service overview

4.3 The Connection Establishment Page

The page concerned with establishing connections is shown in Figure 5. It comprises the four common places and four transitions. The four transitions are used to model the submission and deliv-

ery of TCP service primitives. Each transition is labelled by a primitive it models and its actual name is shown in Table 6. For example, transition *TCP-CONNECT.req* models the submission of the primitive *TCP-CONNECT* request. After it occurs, the client *SAP* changes state from *IDLE* to *OCP*, as indicated by the two arcs connecting place *SAPc* and transition *TCP-CONNECT.req*. At the same time, a *CONNECT* message is put into the queue, modelled by concatenating the current list in place *C_S* with list [*CONNECT*] (arc inscription $q^{\wedge\wedge}[\text{CONNECT}]$). Transition *TCP-CONNECT.ind* models the delivery of primitive *TCP-CONNECT* indication at the server *SAP*. When it occurs, *q* (denoting the tail of the queue in place *C_S*) is returned, indicating that *CONNECT* is removed from the queue. The server *SAP* changes state from *IDLE* into *ICP*. The transitions for the connect response and confirm primitives behave in a similar way, and once they have occurred both interfaces are in the data transfer ready (*DTR*) state.

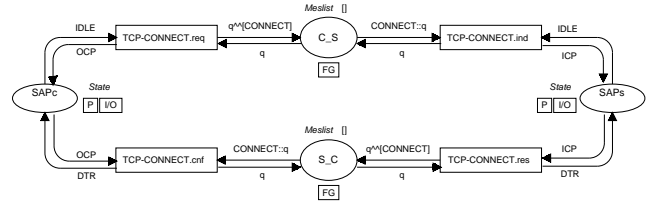


Figure 5: Connection establishment service

4.4 The Data Transfer Page

The data transfer service is modelled in Figure 6 with the four common places and eight transitions. Both the client user and the server user can submit a *TCP-DATA* request, as modelled by two transitions labelled *TCP-DATA.req*. The message *DATA* is sent into the queue after transition *TCP-DATA.req* occurs. The delivery of the *TCP-DATA* indication is modelled by transition *TCP-DATA.ind*. For a data primitive to occur the local interface must be in *DTR* or in either the no data to send (*NDS*) (indication primitives) or no data to receive (*NDR*) (request primitives) state. Note that each *SAP* remains in the same state after a data primitive is submitted or delivered. This is modelled by the bidirectional arc labelled *s*. *TCP-DATA.ind* can only occur if normal data is at the head of the queue. In contrast urgent data has the possibility of being received before any of the normal data ahead of it in the queue. The first situation is modelled by using $\text{DATA} :: q$ as the inscription on the arc from place *C_S* (or *S_C*) to transition *TCP-DATA.ind*. This indicates that for this transition to be enabled *DATA* must be at the head of the queue. The second situation is modelled by using inscription $\text{rm}(\text{UDATA}, q)$ on the arc from transition *TCP-UDATA.ind* to place *C_S* (or *S_C*), and the guard, $\text{member}(\text{UDATA}, q)$. The guard requires that *UDATA* is in the queue. When the *TCP-UDATA.ind* transition occurs, the original list of messages *q* is replaced by $\text{rm}(\text{UDATA}, q)$, the list with the first occurrence of *UDATA* removed from *q*.

4.5 The Connection Release Page

As specified in Tables 3 and 4, the connection release service allows each side to initiate release independently. This is modelled by the eight transitions in Figure 7. The four outside transitions model the invoking of *RELEASE* primitives initiated by the client,

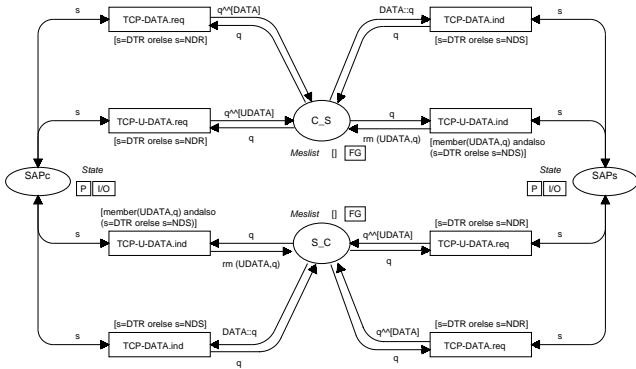


Figure 6: Data transfer service

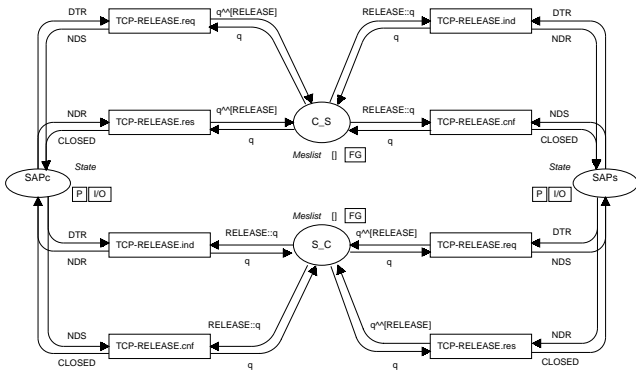


Figure 7: Connection release service

whereas the four inside transitions model that initiated by the server. The state change invoked by the occurrence of a transition reflects that specified for a RELEASE primitive in either Table 3 or 4.

4.6 The Abort Page

The abort service is modelled by six transitions. The two transitions TCP-P_ABORT.ind model the provider initiated abort on each side, while the other transitions model the user initiated abort, which can be invoked at either side. Aborts can occur anytime that the connection is alive (hence in any state other than IDLE or CLOSED). Because no primitive can occur after an abort occurs ABORT is always the last message in the queue. Transition TCP-U_ABORT.ind can occur so long as there is an abort at the end of the queue. This condition is enforced by the guard $hd(rev(q)) = ABORT$ so long as q is not empty. When transition TCP-U_ABORT.ind occurs, the list of messages is removed from the queue and place C.S (or S.C) becomes empty. When any TCP-ABORT primitive occurs, the local SAP changes state to CLOSED.

5 Global Sequences of TCP Service Primitives

To generate the global sequences of TCP service primitives, we use reachability analysis and automata reduction. In this paper, we shall only consider the (finite) sequences associated with connection management. Thus we omit the data transfer service.

The reachability graph, known as an occurrence graph (OG) (Jensen 1997), which is generated by Design/CPN shows that there are 81 markings and 180

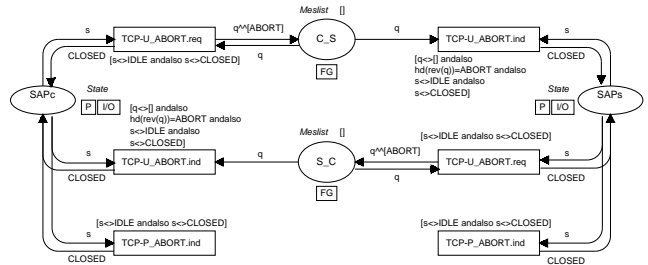


Figure 8: Abort service

transitions for the connection establishment, release and abortion services.

To obtain a minimised representation of the global sequences of service primitives, the OG is considered to be a finite state automaton (FSA). It is converted to a suitable format and entered into the FSM library (AT&T Labs 2002a), which produces the minimised deterministic FSA (which preserves the sequences of primitives - the *service language*). We define halt states in terms of the SAP states. One category is that both SAPs are CLOSED. The other is that the client SAP is CLOSED while the server SAP is IDLE. This corresponds to the situation where a TCP-ABORT primitive occurs at the client SAP after a TCP-CONNECT request is issued. In this case, a TCP-CONNECT indication need not be delivered to the server.

The FSA is minimised using FSM and contains 40 nodes and 147 arcs. Three halt states result from the conditions defined on the SAP states (above), due to different states of the queues. Other statistics are collected using Lextools (AT&T Labs 2002b) and a script language (e.g., Perl (Schwartz & Christiansen 1997)), as summarised in Table 5. There are 288 sequences. The maximum sequence length is 8, while the minimum sequence length is 2. One of the longest sequences is the one shown in Figure 1 without the data transfer primitives (TCP-DATA.req and TCP-DATA.ind). The two shortest sequences are a TCP-CONNECT request followed by a TCP-U-ABORT request or a TCP-P-ABORT indication.

Nodes	Arcs	Halts	Sequences	MaxL	MinL
40	147	3	288	8	2

Table 5: Language statistics

The global sequences of TCP service primitives are shown in Figure 9. Service primitives associated with FSM transition names are shown in Table 6. Each transition is named with four letters. The first letter is the initial of a service primitive name, except for TCP-U-ABORT which is represented by letter A. The last three letters are the abbreviation of a service primitive type. For example, CREQ represents the TCP-CONNECT service primitive of type request. A transition is named in upper case if it models the invoking of a primitive at the client side. It is named in lower case if it models the invoking of a primitive at the server side.

In Figure 9, the longest sequence discussed above can be seen as the path 0,1,2,5,10,16,25,32,39, where the numbers represent the nodes in the FSA. As another example, one of the two shortest sequences is 0,1,3 representing the user aborting the connection immediately after requesting it.

Primitive	Transition Name
TCP-CONNECT.req	CREQ
TCP-CONNECT.ind	cind
TCP-CONNECT.res	crs
TCP-CONNECT.cnf	CCNF
TCP-RELEASE.req	RREQ, rreq
TCP-RELEASE.ind	RIND, rind
TCP-RELEASE.res	RRES, rres
TCP-RELEASE.cnf	RCNF, rcnf
TCP-U-ABORT.req	AREQ, areq
TCP-U-ABORT.ind	AIND, aind
TCP-P-ABORT.ind	PIND, pind

Table 6: The mapping from primitives to transition names

6 Conclusions and Future Work

This paper presents a service specification for the Internet's Transmission Control Protocol (TCP). Our TCP service definition aims to be implementation independent and is therefore defined according to the OSI service conventions. This is in contrast to other attempts that have based their definitions on the TCP/Application Interface definition of RFC 793. We adopt the accepted practice of using service primitives to provide abstract representations of the communication across the TCP/Application Interface, and only define primitives that are involved in end-to-end communication. We thus do not have a service primitive that corresponds to a Passive Open used by TCP servers to indicate a willingness to accept connections, because it only affects the local host. Similarly we do not provide primitives for the Receive and Status calls defined in RFC 793.

Our TCP service definition is based on the International Standard for the OSI Transport Service. It differs from the OSI Transport Service in that the TCP service includes an orderly connection release service. We include the orderly release service based on the OSI Session service's orderly release primitives. However, the TCP service differs from the OSI Transport and Session Service definitions in its handling of connection refusal, which in TCP is dealt with at the provider level (without informing the user), due to what appears to be the intent of the passive open and close interface calls.

We formalise the TCP service using Coloured Petri Nets. This allows us to capture all the global sequences of service primitives, not only the local interface primitive sequences. We demonstrate how to generate the global sequences for the Connection Management primitives, by using reachability analysis and automata reduction techniques, and present the result as a 40 node automaton. This shows that these sequences are as expected and contain sequences varying in length from 2 primitives to 8 primitives.

The service specification presented in this paper captures all of the main features of the service provided by TCP (connection establishment, data transfer, urgent data transfer, orderly release and abort) for the client-server scenario. However, it does not deal with the completely symmetric situation which includes the rare but possible simultaneous opening of connections. We do not see this as a difficult extension, as we shall just allow for the use of all four Connect primitives at each local interface, however, the details of this extension are the subject of future work.

We are interested in the verification of TCP against this service specification. As a first step

we are in the process of creating a comprehensive Coloured Petri Net model of TCP connection management procedures operating over the service provided by the Internet Protocol. Early results of this work have been presented in (Han & Billington 2002). We intend to extend this work to allow the sequences of primitives generated by the TCP model to be compared with the TCP service presented in this paper.

References

- Allman, M., Paxson, V. & Stevens, W. (1999), TCP Congestion Control, RFC 2581, IETF.
- AT&T Labs (2002a), FSM Library. Web site: <http://www.research.att.com/sw/tools/fsm>.
- AT&T Labs (2002b), Lextools. Web site: <http://www.research.att.com/sw/tools/lextools>.
- Billington, J. (1983), 'Abstract Specification of the ISO Transport Service Definition using Labelled Numerical Petri Nets', *Protocol Specification, Testing, and Verification III*, 173–185.
- Braden, R. (1989), Requirements for Internet Hosts — Communication Layers, RFC 1122, IETF.
- Cass, D. E. & Rose, M. T. (1986), ISO Transport Services on Top of the TCP, RFC 983, IETF.
- Floyd, S. & Henderson, T. (1999), The NewReno Modification to TCP's Fast Recovery Algorithm, RFC 2582, IETF.
- Han, B. & Billington, J. (2002), Validating TCP Connection Management, in C. Lakos & R. Esser, eds, 'Proceedings of the Workshop on Software Engineering and Formal Methods, Adelaide, Australia', Vol. 12 of *Conferences in Research and Practice in Information Technology*, pp. 47–55.
- IETF (2002), Internet Engineering Task Force. Web site: <http://www.ietf.org>.
- ITU-T (1993), Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services. ITU-T Recommendation X.210.
- ITU-T (1995a), Information Technology - Open Systems Interconnection - Session Service Definition. ITU-T Recommendation X.215.
- ITU-T (1995b), Information Technology - Open Systems Interconnection - Transport Service Definition. ITU-T Recommendation X.214.
- Jacobson, V., Braden, S. & Borman, D. (1992), TCP Extensions for High Performance, RFC 1323, IETF.
- Jensen, K. (1997), *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1, Basic Concepts*, Monographs in Theoretical Computer Science, 2nd edn, Springer-Verlag, Berlin.
- Murphy, S. L. & Shankar, A. U. (1991), 'Connection Management for the Transport Layer: Service Specification and Protocol Verification', *IEEE Transactions on Communications* **39**(12), 1762–1775.
- Paxson, V. (1999), Known TCP Implementation Problems, RFC 2525, IETF.

- Postel, J. (1981), Transmission Control Protocol, RFC 793, IETF.
- Pouffary, Y. & Young, A. (1997), ISO Transport Service on top of TCP (ITOT), RFC 2126, IETF.
- Rose, M. T. & Cass, D. E. (1987), ISO Transport Service on top of the TCP Version: 3, RFC 1006, IETF.
- Schwartz, R. L. & Christiansen, T. (1997), *Learning Perl*, 2nd edn, O'Reilly&Associates, Sebastopol, CA.
- Smith, M. A. (1996), 'Formal Verification of Communication Protocols', *Formal Description Techniques IX: Theory, Applications, and Tools* pp. 129–144.
- Symons, F. J. W. (1978), Modelling and Analysis of Communication Protocols using Numerical Petri Nets, Ph.D Thesis, University of Essex.
- Tanenbaum, A. S. (1996), *Computer Networks*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
- Ullman, J. (1994), *Elements of ML Programming*, Prentice Hall, Englewood Cliffs, NJ.
- University of Aarhus (1996), *Design/CPN Online*, Web site: <http://www.daimi.au.dk/designCPN>.
- Vissers, C. A. & Logrippo, L. (1986), 'The Importance of the Service Concept in the Design of Data Communications Protocols', *Protocol Specification, Testing, and Verification V*, 3–17.