Abruptly Terminated Connections in TCP — A Verification Example

Ina Schieferdecker
GMD Fokus
Hardenbergplatz 2, 10623 Berlin, Germany
ina@fokus.gmd.de

Abstract

The paper presents the verification of a functional misbehavior in TCP, that is one of the widely used transport protocols used in the Internet. The solution that was developed by I. Heavens is verified for its correctness. A model checking approach is used to verify TCP: the protocol is described in LOTOS and the requirements are given in the modal μ-calculus. The verification uses the CADP tool set for automating the verification process.

1 Introduction

Only recently a functional error in the TCP mechanism for closing connections was recognized [4]. TCP uses a three-way handshake to close a connection with `FIN` messages in an orderly fashion. A special state `Timed_wait` is used in order to ensure, firstly, that all `FIN` messages are reliably acknowledged for the completion of the three-way handshake, and, secondly, that all TCP segments, generated in either direction during the lifetime of the connection, are reliably delivered and drained from the network before initiation of a new incarnation[1] of the connection.

On the other hand, connections can be closed abruptly with `RST` messages, so that the connection state of the peer entities on transmission or reception is immediately removed. However, no equivalent mechanism to `Timed_wait` exists for connections terminated by the transmission of a `RST` message. Hence, TCP contains the possibility of erroneous acceptance of old segments from `RST`-terminated connections, which are for example caused by user aborts or by the reception of data after half-duplex close.

The problem was whether it would have been possible to detect the misbehavior of TCP by applying formal verification without any use of the knowledge about the error. Could the error have been avoided, if formal verification was used before any implementation of TCP? We decided to solve the problem in the framework of functional behavior specification of the protocol, temporal logic specification of the correctness requirements, and verification by the use of model checking algorithms. The functional behavior of TCP is specified in LOTOS [5], while the correctness requirements are specified in the modal μ-calculus [8].

The paper is structured as follows. Before describing the misbehavior for abruptly terminated connections, the main features of TCP are shortly discussed. After a short overview on used formalisms, the LOTOS specifications of TCP and of the proposed solution are presented. The verification that uses the `evaluator` tool of the Caesar/Aldebaran Distribution Package (CADP) [3] is considered next. Conclusions finish the paper.

2 The Transmission Control Protocol

The Transmission Control Protocol (TCP) [2] as part of the TCP/IP protocol suite is beside the User Datagram Protocol (UDP) the most widely used transport protocol in the Internet. It has been developed in the late 1970s and was standardized by the Internet Engineering Task Force in 1981 [6]. Several corrections and improvements of TCP [1] lead to robust and performant implementations of TCP, among which the BSD4.4. implementation of TCP should be mentioned as a well-known representative [7].

TCP provides a reliable data transmission service, namely, the in-order delivery of a stream of bytes. The stream of bytes is separated into segments which are transferred to the other side. It is a connection-oriented and full duplex transport protocol, that uses an application to application addressing scheme. A connection is identified by the addresses of both sides, where an address consists of the IP and the port number. The reliability of the data transfer is achieved by the use of

- sequence numbers for every byte in the stream,

[1] TCP uses the notion incarnation to denote new instances of a connection, i.e. having the same socket pair of sending and receiving address.

- positive acknowledgments that indicate the next data to be received,

- retransmission of segments that are not acknowledged within a timeout period, and

- round trip delay estimation for adapting the timeout periods to the actual load situation in the network.

TCP uses a sliding window mechanism to prevent the sender from overloading the receiver (flow control). Slow start and congestion avoidance schemes are used for congestion control, so that the sender does not overload the network. There are several other features of TCP such as the negotiation of the maximal segment size (MSS), the selection of the initial sequence number (ISN), Karn's algorithm to avoid the silly window syndrome, fast retransmission and fast recovery, etc. Only the basic features of TCP are relevant to the problems of abruptly terminated connections, so that the others are not considered here. For further detail on TCP please refer to [7]. TCP offers the following user calls to applications:

TCP supports passive and active `OPEN` calls. A passive open makes a TCP entity ready to accept requests for connection setups from remote sides, while an active open initiates the segment exchange that is needed for connection establishment, i.e. the connection setup is actively pursued.

The user data are given to the sending TCP entity by `SEND` calls and are transferred as sequences of octets. TCP must recover from lost, duplicated, and re-ordered data. This is achieved by assigning sequence numbers to any octet in the stream and by positive acknowledgments from the remote side.

Incoming data are delivered to the user of the receiving side, when using a `RECEIVE` call.

`CLOSE` is the user call for terminating a connection in normal situations.

`ABORT` means the immediate termination of a connection without any prevention for lost connection termination indications and clearing old data from the network.

TCP entities communicate by means of segments. A segment contains, beside other things, the sequence and acknowledgment number, several control bits, and the data. The control bits are used to mark the specific meaning of a segment. For the investigation in this paper the following control bits are essential:

`ACK` to indicate an acknowledgment segment. Acknowledgments are used to inform the sender about successful receipt of data and to indicate the sequence number of the next awaited data.

`RESET` (reset the connection) to indicate the abrupt termination of a connection.

`SYN` to indicate synchronize segments for the exchange of information on initial sequence numbers during connection setup.

`FIN` to indicate that no more data are sent from the sending side. `FIN`s are used when connections are normally terminated.

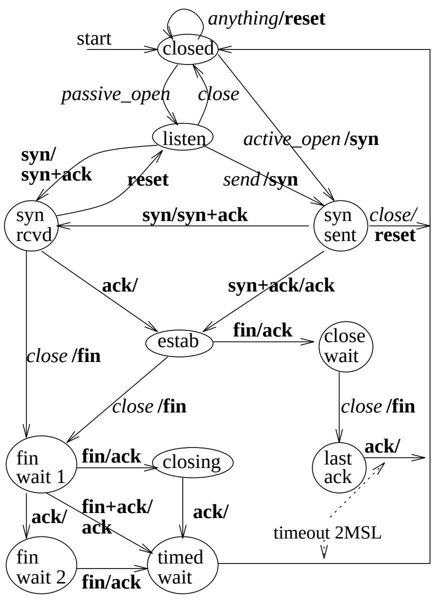

Figure 1: The finite state machine of TCP

The behavior of a TCP connection for connection setup and termination is represented in Figure 1[2]. The figure uses a finite state machine representation of the behavior. States and transitions are represented by circles and edges, respectively. A transition is marked with the input and the output event. There are transitions without output event. In order to distinguish between a user call and a segment, user calls are written in italics. TCP has eleven states with the following meanings

`Closed` is the idle state where no connection exists at all.

`Listen` The entity waits for a connection request from any remote side.

[2]Please note, that this automaton only contains the main control flow of TCP and by far not the complete behavior. For example, the reaction on `ABORT` user calls is only partially represented and the data transfer phase is not shown at all.

`Syn_sent` is the waiting for a matching connection request after having sent its own connection request.

`Syn_rcvd` represents waiting for the acknowledgment of its connection request after it has both received and sent a connection request.

`Estab` is the state of the data transfer phase. Data can be sent to the remote side and received data can be delivered to the user of the TCP entity.

`Fin_wait_1` represents waiting for a request on connection clearing from the remote side or for the acknowledgment of the previously sent request on connection clearing.

`Fin_wait_2` is the waiting state for a request on connection clearing from the remote side.

`Close_wait` represents waiting for a termination request from the local user.

`Closing` awaits the acknowledgment for a previously sent request on connection clearing.

`Timed_wait` is the state of waiting long enough, so that the remote side eventually receive the acknowledgment of its termination request and so that all data of the connection have left the network.

`Last_ack` is similar to the `Closing` state, but is used after a `Close_wait` state.

3 Old Data Acceptances Caused by Abruptly Terminated Connections

Before describing the problems with abruptly terminated connections, the normal connection establishment and connection termination in TCP is considered.

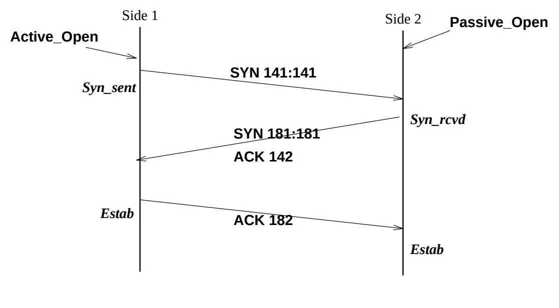

Figure 2: Connection establishment

During normal connection establishment (Figure 2), the initiating side of the call sends a `SYN` segment with its initial sequence number to the remote side. The called side answers with acknowledging the initial sequence number of the call initiator and sends its own initial sequence number in a `SYN` segment back. In the case that the calling side receives an acknowledgment for its `SYN` segment, it transfers to the `Estab` state. Finally, the called side awaits the acknowledgment of its `SYN` segment before entering the `Estab` state. Hence, three segments are used to set up a connection, what is also called three-way handshake. Since both of the `SYN` segments have to be explicitly acknowledged by the other side, it is ensured that both entities have the correct information on the sequence numbers of those segments that the remote side will eventually send.

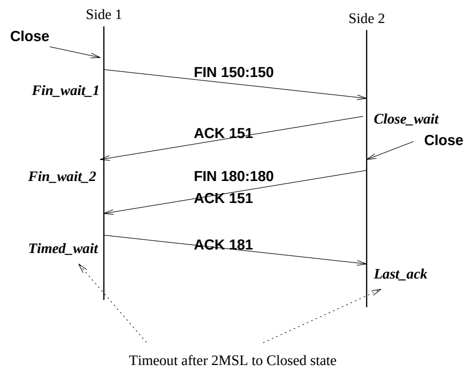

Figure 3: Connection termination

In the course of normal connection clearing (Figure 3), a modified three-way handshake mechanism is used: after a user `CLOSE` call, the entity to be closed sends a `FIN` segment to the remote side. That side acknowledges the incoming `FIN` segment immediately and delays until its local user also closes the connection. Afterwards, a `FIN_ACK` segment is send to the remote side, from which the last acknowledgment for the `FIN` segment is awaited. Similar to the connection setup, the explicit acknowledgment of the `FIN` segment ensures that both sides are aware of the closing connection. In the `Timed_wait` and `Last_ack` state, an idle period of two times the maximal segment lifetime (`2MSL`) is used in order to ensure that all data from the closing connection diminish from the network.

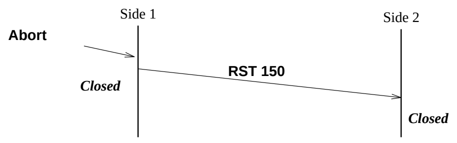

Figure 4: Connection abort

In contrast to the secure connection termination in normal situations, a connection can be abruptly terminated by issuing an ABORT call (Figure 4). Although a RESET segment is sent to the remote side, no acknowledgment is awaited. This implies that the remote side may not be aware of the connection closing at the remote side (in the case that the RST segment gets lost). In addition, old data may still be in transit to the other side, since there is no idle period of 2MSL. In particular, this means that the sequence numbers of old data may overrun the sequence numbers of a new incarnation of this connection.

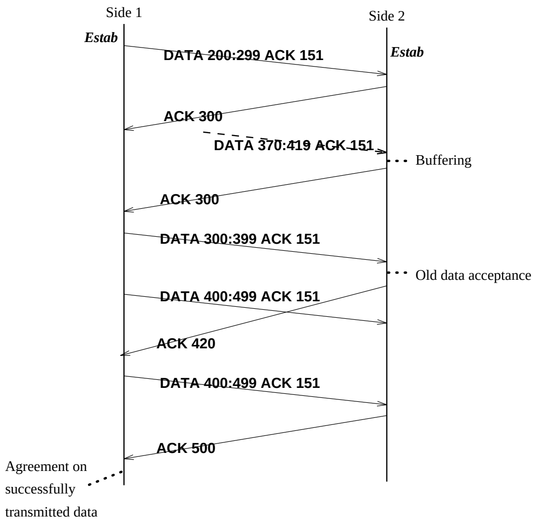

Figure 5: Old data acceptance

Figure 5 shows a scenario where old data are accepted in the new incarnation. Side 1 serves as the data source, while Side 2 is the data sink. Let us assume that there are still old data in the network and that the sequence numbers of the new incarnation are overrun by the sequence numbers of old data. The first two lines show normal data (DATA 200..299) and acknowledgment (ACK 300) of the new incarnation (after the three-way handshake for setting up the new incarnation has been completed). Next, old data (DATA 370..419) arrive at Side 2, which fall into the current receiving window of Side 2. Therefore they are queued into the re-assembly buffer at Side 2, since their sequence numbers are not the expected ones. An acknowledgment is generated for the next expected octet (ACK 300), so that Side 1 does not detect an acknowledgment for unsent data. Side 1 continuously sends further data to the other side (DATA 300..399, 400..499), since its sending window is open. Once, Side 1 receives the missing data (DATA 300..369) as part of the second data segment sent by Side 1, it accepts old data and offers it to its user. It acknowledges data up to 420 to the sending side. Side 1 interprets this acknowledgment as a partial acknowledgment for its third data segment. Since the timer does not expire and the sending window is still open, it continues sending. Finally, it receives the acknowledgment for all sent data (ACK 500), so that neither Side 1 nor Side 2 detected the faulty acceptance of old data.

Beside the theoretical possibilities of old data acceptances in the course of connection abort and new connection incarnation, it has to be investigated whether the time settings and the mechanism for the selection of the initial sequence number (ISN) of a connection do not exclude this misbehavior of TCP.

In particular, if we assume a high speed network with large throughput and long propagation delays, one can consider a scenario of one continuously sending side and of a receiver that abruptly terminates the connection and re-establishes a new incarnation of the connection immediately. Then, the sending side will continue to send, although the RST segment is on the way. Therefore it will generate a whole bunch of old data, which may overrun the sequence numbers of the new incarnation.

According to the TCP standard [6], TCP uses a 32-bit counter for the ISN selection mechanism, which is incremented every 4 microseconds. That means 250000 increments per second. If we assume a TCP connection in a high speed network with maximal segment size (MSS) of 1500 Byte and a sending window size of 170 segments, the sender can transmit data segments that use this number space. In this scenario, there is a non-zero probability for old data acceptances if the connection is re-established immediately. However, such a transmission capacity requires a highspeed network in the hundreds of MBit/s range.

Also, the ISN selection mechanisms in current TCP implementations are comparable to the TCP standard. For example, the TCP implementation of 4.4. BSD UNIX uses a slower counter (the counter is incremented by 64000 every half-second, i.e. the counter is incremented every 8 microseconds). However, in addition, the counter is incremented by 64000 each time a connection is established.

Hence, it is unlikely to observe this misbehavior of TCP in current networks. Nonetheless, [4] showed old data acceptance phenomena by a simple (although unrealistic) experiment consisting of a client application that continuously sends data, and of a server application that aborts the connection with any arrival of a data segment and that reconnects immediately afterwards. This continuous repetition of pushing old data into the network and of establishing new incarnations leads eventually to the above described old data acceptances.

In order to solve the problem, one can ask (1) for the adaptation of the ISN selection mechanism to the needs of future high speed networks, so that it is fast enough to exclude segment number overruns by old data. On the other hand, (2) one can deploy a general solution, so that the functional behavior of TCP is free of misbehavior independently of the ISN selection mechanism.

The first alternative has the advantage of letting the functional behavior of TCP as it is, what is important for all the applications that run over TCP. The disadvantage is the need to correct the timing of TCP whenever a new generation of communication networks is used. While the second alternative solves the latter problem, the functional correction of TCP cannot be undertaken without problems. In particular, this is true for the large number of installed TCP implementations and for the large number of applications that utilize TCP. Care has to be taken of backward compatibility issues. In order to ensure backward compatibility, options for "classical" TCP and its new variant should be offered, what would complicate the protocol.

The presented work will not discuss pros and cons of both alternatives further. Rather, the main goal was to formally verify the misbehavior of TCP as well as the proposed functional solution to the problem [4].

4 The Solution to Prevent TCP from Old Data Acceptances

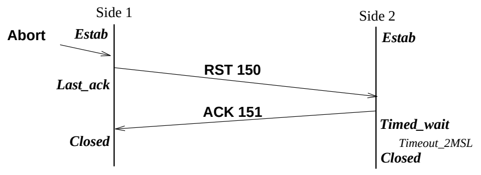

Figure 6: Modified connection abort

In order to exclude the above described old data acceptances after connection aborts, [4] proposes to apply a two-way closing mechanism to abruptly terminated connections. This mechanism ensures that RST segments are reliably delivered to the remote side (Figure 6). Furthermore, also in the case of aborts, the Timed_wait state is used to wait long enough for old data diminishing from the network.

An ABORT call (see Figure 7) in a state that is different to the Listen and Syn_sent state causes a RST segment with explicit sequence number (so that it can be retransmitted in the case of loss) sent to the remote side. Afterwards, the TCP entity transfers to the Last_ack state where the acknowledgment for the previously sent RST segment is awaited. If the ACK segment arrives, the connection is closed.

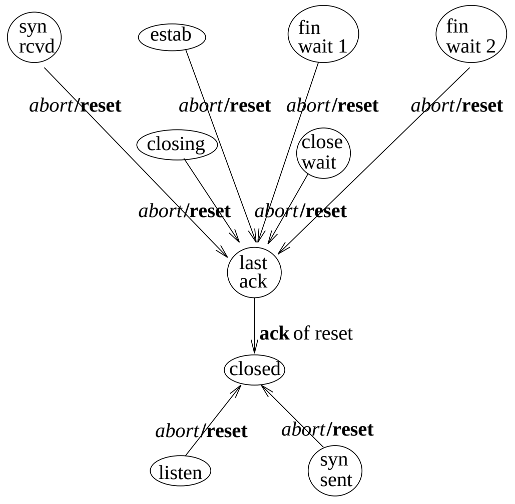

Figure 7: Modified receipt of ABORT

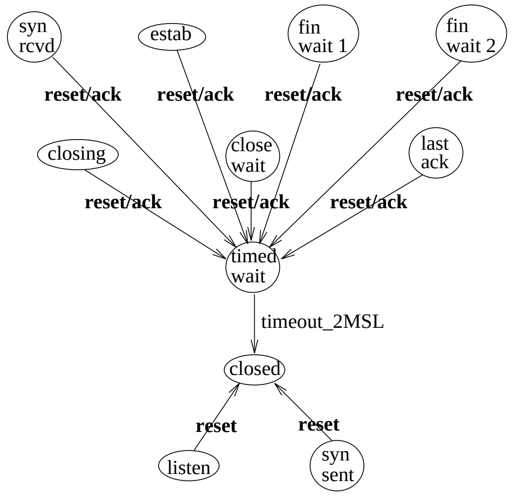

Figure 8: Modified receipt of RESET

Furthermore, every arriving RST segment has to be explicitly acknowledged in a state that is different to the Listen and Syn_sent state (Figure 8). In these two states upon arrival of a RST segment, an immediate transition to the Closed state is used. After acknowledging the RST segments in the other states, Timed_wait state is entered for the waiting period of 2MSL.

5 An Overview on Used Formalisms

5.1 An Overview on LOTOS

The specification of TCP has been developed in LOTOS — the Language Of Temporal Ordering Specification [5], what is one of the standardized formal description techniques. It has been developed by ISO for the unambiguous definition of the functional behavior of information processing systems. It is based on process-algebraic calculi for the description of behavior, in particular on Milner's CCS — the Calculus of Communicating Systems — and on Hoare's CSP — the calculus of Communicating Sequential Processes. Data dependencies are described in the algebraic data type language ACT ONE.

A LOTOS specification defines the system behavior as the temporal order of externally visible events. Each event is an occurrence of an action and is associated with a gate, namely the gate at which the event takes place. The set of gates constitutes the system interface. The basic notions of LOTOS are *actions* that represent atomic and instantaneous system functionalities, and basic *processes* (stop and exit) that represent deadlock and termination, respectively. The occurrence of an action is called *event*. Actions and processes can be composed by a number of operators such as action prefixing or parallel composition in order to build complex behavior expressions:

- The stop process "`stop`" denotes deadlock, i.e. a process that is unable to execute anything.
- The exit process "`exit` ($\mathtt{e}_1 \ldots \mathtt{e}_n$)" denotes successful termination, where value may be passed to subsequent processes.
- An action prefix "`g` $\mathtt{e}_1 \ldots \mathtt{e}_n$ `[SP]; Q`" represents an observable action `g` with value offering $\mathtt{e}_1 \ldots \mathtt{e}_n$ to the environment, provided that the communication guard `SP` evaluates to true. The action occurrence is followed by behavior `Q`.
- An internal action prefix "`i; Q`" means occurrence of the internal action `i` followed by behavior `Q`.
- A choice expression "$\mathtt{Q}_1$ `[]` $\mathtt{Q}_2$" denotes choice between alternative behaviors $\mathtt{Q}_1$ and $\mathtt{Q}_2$. Only one of them is selected for further execution.
- A parallel composition "$\mathtt{Q}_1$ `|[`$\mathtt{g}_1 \ldots \mathtt{g}_n$`]|` $\mathtt{Q}_2$" represents the parallel execution of behaviors $\mathtt{Q}_1$ and $\mathtt{Q}_2$, which have to synchronize in the gates $\mathtt{g}_1 \ldots \mathtt{g}_n$. LOTOS uses synchronous communication (with rendez-vous).
- The hiding operator "`hide` $\mathtt{g}_1 \ldots \mathtt{g}_n$ `in Q`" makes $\mathtt{g}_1 \ldots \mathtt{g}_n$ invisible and unaccessible from outside.
- An enabling "$\mathtt{Q}_1$ `>> accept` $\mathtt{x}_1$:$\mathtt{s}_1$ $\ldots \mathtt{x}_n$:$\mathtt{s}_n$ `in` $\mathtt{Q}_2$" denotes the sequential composition of $\mathtt{Q}_1$ and $\mathtt{Q}_2$, where values may be passed to $\mathtt{Q}_2$ after termination of $\mathtt{Q}_1$.
- The disabling expression "$\mathtt{Q}_1$ `[>` $\mathtt{Q}_2$" denotes the possibility of interrupting $\mathtt{Q}_1$ by $\mathtt{Q}_2$.
- A process is instantiated by "`P[`$\mathtt{g}_1 \ldots \mathtt{g}_n$`]` ($\mathtt{e}_1, \ldots, \mathtt{e}_m$)", which denotes the execution of the process behavior of `P` with actual gates $\mathtt{g}_1 \ldots \mathtt{g}_n$ and actual parameters $\mathtt{e}_1, \ldots \mathtt{e}_m$. In particular, the recursive re-instantiation of processes allows the specification of infinite, cyclic behavior.
- A guard expression "`[SP]-> Q`" denotes the execution of `Q`, if the guard `SP` evaluates to true.
- A value declaration "`let` $\mathtt{x}_1$:$\mathtt{s}_1$=$\mathtt{e}_1$ $\ldots$ $\mathtt{x}_n$:$\mathtt{s}_n$=$\mathtt{e}_n$ `in Q`" binds in behavior `Q` the values $\mathtt{e}_1 \ldots \mathtt{e}_n$ to the variables $\mathtt{x}_1 \ldots \mathtt{x}_n$, resp.

A system is normally represented by a composition of several processes. The interface of a process is defined by a set of gates that identify the externally visible and accessible actions and in which the environment of the process may synchronize. Processes can built up process hierarchies, what is useful to structure the specification.

5.2 An Overview on the Modal μ-Calculus

Modal logics are used to describe behavioral properties in terms of capabilities of processes [8]. Formulas are built from boolean connectives, modal operators $[K]$ ("box K") and $\langle K\rangle$ ("diamond K"), and fixed point operators νZ and μZ, where Z denotes a propositional variable and K a set of actions. The formulae of the logic[3] are

$$\begin{aligned}\Phi ::= {} & \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \\ & \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \\ & [K] \mid \langle K\rangle \mid \\ & \nu Z.\Phi \mid \mu Z.\Phi.\end{aligned}$$

The meaning is as follows. `tt` and `ff` denote true and false, resp. A process has the property $[K]\Phi$ if after execution of any action in K each resultant process has property Φ. The property $\langle K\rangle\Phi$ expresses the capability of executing an action in K, so that the resultant process has property Φ.

$\nu Z.\Phi$ denotes the greatest fix-point of the equation $Z \overset{def}{=} \Phi$, while $\mu Z.\Phi$ is the least fix-point of this equation. For example, $\nu Z.\langle a\rangle Z$ expresses the capability for performing action a forever. Another example is $\mu Z.\Phi \vee \langle a\rangle Z$ that expresses the property of performing action a until Φ holds. This least fix-point also includes the possibility of performing action a forever without Φ ever becoming true. Last but not least, please observe that `tt` expresses the same as

[3]Please note, that this logic is not minimal, but reflects the main features of the modal μ-calculus.

the greatest fix-point of the equation $Z \overset{def}{=} Z$ and ff expresses its least fix-point.

The evaluator of CADP supports shortcuts to ease the formulation of properties: $\langle * \rangle \Phi$ denotes the property of performing an action from the complete set of actions of a process, so that the resultant process has property Φ, while $[*]\Phi$ denotes the property that after performing any action each resultant process has property Φ. $ALL\Phi \overset{def}{=} \nu Z.(\Phi \wedge [*]Z)$ denotes the property that Φ is true for all processes that can be reached, i.e. Φ is always satisfied. $POT\Phi \overset{def}{=} \mu Z.(\Phi \vee \langle * \rangle Z)$ denotes the property that there are some resultant processes that have property Φ. $WU\Phi_1\Phi_2 \overset{def}{=} \nu Z.((\Phi_1 \wedge [*]Z) \vee \Phi_2)$ denotes weak until, i.e. it denotes the property that the resultant processes have property Φ_1 as long as they do not have property Φ_2, even forever if need be.

6 The Specification of TCP

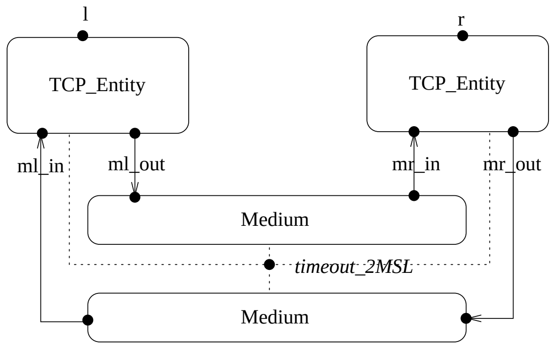

Figure 9: The structure of the TCP specification

The TCP specification is the result of taking into account the original TCP specification [6], the TCP corrections given in [1], and the reference implementation that is described in [7]. The specification reflects the functional behavior of TCP without dynamically adjusted window sizes, slow start and congestion avoidance, fast retransmission and fast recovery, buffering at the receiving and sending side, checksums, etc. The specification contains 300 lines of code.

The structure of the specification is presented in Figure 9. The specification consists of a TCP entity at each side. Both entities use a reliable transmission medium with data reordering. The corresponding specification of the main behavior is given in Figure 10.

The specification uses enumerations for the definition of user calls, segments, and states. Exemplarily, the definitions for user calls and segments are given in Figure 11. The representation of user calls, segments, and states is in particular useful to reduce the size of the transition systems (as a number of bytes), which are generated by caesar and aldebaran (see below).

```
specification TCP [l,r]: noexit
...
behavior
hide ml_out, ml_in, mr_out, mr_in, timeout_2MSL in
(  TCP_entity_closed[l,ml_in,ml_out,timeout_2MSL](1,0,0,0)
 |||
   TCP_entity_closed[r,mr_in,mr_out,timeout_2MSL](1,0,0,0))
|[ml_out, ml_in, mr_out, mr_in, timeout_2MSL]|
(  Medium[ml_out,mr_in,timeout_2MSL]
 |[TIMEOUT_2MSL]|
   Medium[mr_out,ml_in,timeout_2MSL] )
where ... endspec (*TCP*)
```

Figure 10: The system behavior

```
type UMessage is NaturalNumber renamedby
 sortnames UMessage for Nat
 opnnames PASSIVE_OPEN for 0
          ACTIVE_OPEN for 1
          SEND for 2
          RECEIVE for 3
          CLOSE for 4
          ABORT for 5
          STATUS for 6
endtype
type Segment is NaturalNumber renamedby
 sortnames Segment for Nat
 opnnames RESET for 0
          SYN for 1
          SYN_ACK for 2
          FIN for 3
          FIN_ACK for 4
          ACK for 5
          DATA for 6
endtype
```

Figure 11: The user call and segment definition

The *transmission medium* is given in Figure 12. The external gates of the transmission medium are

- in for incoming segments,
- out for outgoing segments, and
- timeout_2MSL for clearing the medium from any segment that is still in transit.

The medium consists of two independent routes between the in and the out gate. A route (process One_Buffer) works like a reliable buffer of size 1, where segments cannot be lost. Since two routes exist, (1) a segment can be held infinitely long in a route without delivering it at the out gate and (2) segments

```
   process Medium[in,out,timeout_2MSL]: noexit:=
2   ( One_Buffer[in,out] ||| One_Buffer[in,out] )
    [>
4   timeout_2MSL; Medium[in,out,timeout_2MSL]
   where process One_Buffer[in,out]: noexit:=
6             in ?s: Segment [s eq RESET];
              out !s; One_Buffer[in,out]
8         [] in ?s: Segment ?n: Nat
               [(s eq ACK) or (s eq SYN) or
10             (s eq FIN) or (s eq DATA)];
              out !s !n; One_Buffer[in,out]
12        [] in ?s: Segment ?n: Nat ?m: Nat
               [(s eq SYN_ACK) or (s eq FIN_ACK)];
14            out !s !n !m; One_Buffer[in,out]
    endproc (*One_Buffer*)
16 endproc (*Medium*)
```

Figure 12: The transmission medium

can be reordered. Both characteristics are essential to observe the misbehavior of TCP for abruptly terminated connections.

The `timeout_2MSL` gate has been introduced to describe the semantics of the `Timed_wait` state. The synchronization in `timeout_2MSL` is used to empty the medium completely. This is specified by the disabling operator (line 3, 4) on `timeout_2MSL` and the re-instantiation of process `Medium` afterwards.

A *TCP entity* is described by a set of processes, where each process represents the behavior in a specific state. In any state, the entity is ready to accept calls from the user or segments from the remote side that are relevant to the current state. Transitions to other states are reflected by the instantiation of the appropriate process (with actualized parameter settings). If the TCP entity transfers to a state that is different to the current one, a status indication containing the next state is given to the environment in order to ease the validation/verification of the specification.

The interface of a TCP entity consists of

- `u` — the gate to the TCP user for user calls,
- `m_in` — the gate to the underlying transmission medium for incoming segments, and
- `m_out` — the gate to the underlying transmission medium for outgoing segments.

The parameters of the TCP entity are

- `no` — the number of the instance of a connection,
- `sn` — the sequence number of the next segment to be sent,
- `rn` — the sequence number of the next segment to be received,
- `ab` — the number of `ABORT` user calls for a connection,
- `sp` — the number of sent data packets, and
- `rp` — the number of received data packets.

For simplification, we assumed a sending window of infinite length and a receiving window of size 1. That means, that the entity is always able to send data, but accepts only that incoming data with a sequence number identical to `rn`. In order to reduce the size of the state space, the following restrictions were assumed in addition: at most two instances of a connection may be established, at most one `ABORT` user call is allowed per connection and per entity, and finally, at most one data packet can be sent or received per connection. The last assumption is in particular needed to adapt a compositional approach to the generation of the state space.

```
process TCP_entity_estab[u,m_in,m_out,timeout_2MSL]
(no,sn,rn,ab,sp,rp: Nat): noexit:=
   u !CLOSE;
       m_out !FIN !sn; u !STATUS !Fin_wait_1;
       TCP_entity_fin_wait_1[..](no,sn+1,rn,ab)
[] ([ab lt 1] -> u !ABORT;
       m_out !RESET; u !STATUS !Closed;
       TCP_entity_closed[..](2,0,0,0))
[] ([sp lt 1] -> u !SEND;
       m_out !DATA !sn;
       TCP_entity_estab[..](no,sn+1,rn,ab,sp+1,rp))
[] m_in !DATA !rn [rp lt 1];
       u !RECEIVE; m_out !ACK !rn+1;
       TCP_entity_estab[..](no,sn,rn+1,ab,sp,rp+1)
[] m_in !ACK !(sn-1);
       TCP_entity_estab[..](no,sn,rn,ab,sp,rp)
[] m_in !FIN !rn;
       m_out !ACK !rn+1; u !STATUS !Close_wait;
       TCP_entity_close_wait[..](no,sn,rn+1,ab)
[] m_in !RESET;
       u !STATUS !Closed;
       TCP_entity_closed[..](2,0,0,0)
endproc (*TCP_entity_estab*)
```

Figure 13: The TCP entity in `Estab` state

As a representative for the state processes, the process for the `Estab` state is given in Figure 13 . Every state process is described by a choice of alternatives on user calls and incoming segments. An alternative in the choice expression describes one transition in the current state. It has an input, possibly an output, and ends in the final state.

For example, if the user issues a `CLOSE` call, the entity sends a `FIN` segment to the remote side with the current send sequence number `sn`. Then it enters the `Fin_wait_1` state by instantiating the `TCP_entity_fin_wait_1` process and by incrementing the send sequence number by `1`. If a data segment arrives (`m_in !DATA !rn`), its sequence number is checked to coincide with the sequence number to be received next (rn). The TCP entity remains in Estab state and re-instantiates with the actualized parameters. Since the transmission medium is assumed to be error-free, no timeout mechanism for the retransmission of lost data is needed.

7 The Specification of Modified TCP

The corrections that are needed to represent the above presented modifications to TCP are rather simple. The appropriate transitions for ABORT calls (line 2–4) are modified to transitions to the Last_ack state (line 5–7):

```
  ...
2 [] ([ab lt 1] -> u !ABORT;
       m_out !RESET; u !STATUS !Closed;
4      TCP_entity_closed[..](2,0,0,0)) ...
  => ...
6   [] ([ab lt 1] -> u !ABORT;
          m_out !RESET !sn; u !STATUS !Last_ack;
8         TCP_entity_last_ack[..](no,sn+1,rn,ab+1)) ...
```

Likewise, the appropriate transition for incoming RST segments (line 2–4) are modified to transitions to the Timed_wait state (line 5–7):

```
  ...
2 [] m_in !RESET;
       u !STATUS !Closed;
4      TCP_entity_closed[..](2,0,0,0) ...
  => ...
6   [] m_in !RESET !rn;
          m_out !ACK !rn+1; u !STATUS !Timed_wait;
8         TCP_entity_timed_wait[..](no,sn,rn+1,ab) ...
```

8 The Verification

The verification used the evaluator tool of CADP [3] for model checking the requirement that during the lifetime of a connection instance data is not received before it has been sent. Before applying the evaluator, caesar, aldebaran, and bcg_open were used to generate the labelled transition system (LTS) of the TCP specification and its modified version.

Instead of verifying the TCP specification presented above we had to use a specification where the data part has been eliminated. This simplification was needed due to the state space exploration problem: a single entity (with data) has a labelled transition system with over 1,1 Mio states and 1,6 Mio transitions (the file containing the LTS has a size of 48MB), so that it was not possible to minimize even one entity, not to mention the complete system[4].

[4] A Sun SPARCstation 20 with two processors, 60 MHz, and 50MB main memory has been used.

Also, the try to validate the correctness requirements on the fly without explicitly constructing the complete state space was unsuccessfully interrupted after four days.

Of course, care has to be taken when eliminating the data part. In particular, the use of sequence numbers is essential for the correct behavior of TCP. Due to the assumptions we made in the TCP specification, (1) data is not lost, and (2) only two segments can be re-ordered on the transit between both entities. However, this can only lead to additional deadlocks, since segments may arrive in states where they are not expected.

The correctness requirements are given by three temporal formulae:

- There is the possibility to establish a connection[5]:

```
POT( <"L !Status !Estab">T
     and
     <"R !Status !Estab">T )
```

- There is the possibility to receive data:

```
POT ( <"L !Receive">T
      or
      <"R !Receive">T )
```

- It is always the case that data is not received before it is sent (with respect to the current instance of a connection):

```
ALL( ["R !Status !Closed"]
     ( WU["R !Receive"]F<"L !Send">T ) )
```

The verification used a compositional approach by generating individually the labelled transition systems of the TCP entity and the medium, before combining them to the complete system[6]. That does not only reduce the execution time for the whole verification process, but gives also more insights into the size of each component. Most importantly, minimizing the LTSs for each component individually reduces the size of the complete LTS (before its minimization) drastically. Please find below the size for an entity, a medium, and the system before and after minimization:

[5] The evaluator had to use in fact POT(<"L !6 !4">T and <"R !6 !4">T) since the user calls, segments, and states are represented by natural numbers.

[6] This approach is supported by the aldebaran tool of CADP: it offers the possibility to generate a LTS from a network of communicating LTSs, which were previously generated.

Component	Before Minimization		
	Transitions	States	Bytes
Medium	13821	901	33K
TCP			
Entity	109	76	2K
System	403964	160931	8,5MB
Modified TCP			
Entity	105	73	2K
System	110992	45083	2,2MB
Component	After Minimization		
	Transitions	States	Bytes
Medium	286	66	5,5K
TCP			
Entity	67	34	1K
System	292790	100261	5,9MB
Modified TCP			
Entity	69	37	1K
System	62127	21871	1,2MB

The generation of the evaluator took for both versions appr. 5 min.

The requirements were checked for their fulfillment as follows:

System	req1	req2	req3
TCP	20s (true)	20s (true)	2:10min (false)
Modified TCP	14s (true)	11s (true)	53s (true)

In addition to the model checking with the `evaluator`, the `exhibitor` was used to find further scenarios (see Figure 5) where old data are accepted. Such scenarios exist not only for aborts in `Estab` state but also in `Fin_wait_1`,`Fin_wait_2`, `Closing`, and `Close_wait` state.

9 Conclusions

During the specifications phase some bugs in the original TCP specification [6] were found, whose corrections are however contained in [1]. In addition, some ambiguities were found that could only be solved when considering the reference implementation described in [7].

The original goal of this verification experiment has only been partially reached: The complete TCP specification could not be analyzed due to its size. Therefore a number of assumptions were made to reduce the size of the state space. These assumptions had to take into account the knowledge about the misbehavior in order not to exclude the system executions of interest. Nonetheless, it showed that tools for automatic verification are powerful enough to analyze problems of practical needs.

The study on TCP will be continued in order to investigate other interesting properties of TCP. In particular, TCP is a good candidate to explore methods for timed verification, since the protocol makes intensive use of timers in order to exclude functional misbehavior.

References

[1] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Oct 1989.

[2] D.E. Comer. *Internetworking with TCP/IP, Vol. 1 - Principles, Protocols and Architectures.* Prentice Hall Inc., 1995.

[3] H. Garavel. *CADP — The Caesar/Aldebaran Distribution Package.* CNRS/IMAG and INRIA, 1995.

[4] I. Heavens. Problems with TCP Connections Terminated by RSTs or Timers. Internet Draft, Nov 1995.

[5] ISO/IEC, Geneva, Switzerland. *IS 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, 1989.

[6] J. Postel. Transmission Control Protocol, DARPA Internet Program, Protocol specification. RFC 793, Sept 1981.

[7] W.R. Stevens. *TCP/IP Illustrated. Vol. 1 - The Protocols.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[8] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Computer Science, Vol. 2*, pages 477–564. Oxford University Press, 1992.