vrije Universiteit *amsterdam*

Vrije Universiteit Amsterdam

Master Thesis

Formal specification and verification of TCP extended with the Window Scale Option

Author:
Lars Lockefeer

Supervisors:
Wan Fokkink
David M. Williams

July 20, 2013

Abstract

The Transmission Control Protocol (TCP) aims to provide a reliable transport service between two parties that communicate over a possibly faulty network. The responsibilities of TCP can roughly be divided into two categories: connection management and data transmission. Connection management sets up the connections, manages the byte streams and their corresponding states and ensures that connections are closed in a safe manner. Data transmission involves the transfer of segments from the sender to the receiver.

The specification of TCP is far from formal and very complex. The original protocol was only specified in natural language in RFC 793. As TCP was implemented on a large scale, several ambiguities and other issues surfaced. RFC 1122 clarified some of these ambiguities and proposed solutions for many issues. Moreover, the networks that TCP operates over have changed remarkably over the years, and have now reached a bandwidth and speed that had never been envisioned at the time that the protocol was originally defined. To ensure reliable and efficient operation of TCP over these modern networks, RFC 1323 proposes the Window Scale Option that enables the use of windows larger than 2^{16}.

For complex systems, formal specification and verification can be used to manage their complexity. The approach is twofold: first, the system is modelled using formal techniques that have mathematical underpinnings, to obtain a specification. To this specification, verification techniques are then applied to verify the correctness of the system.

In this work, we give a formal specification of TCP extended with the Window Scale Option using the process algebra μCRL. Due to the complexity of the protocol, our specification focuses on the data transfer phase and connection teardown. From this specification, we obtain a model of unidirectional data transfer and show that its external behaviour is branching bisimilar to a FIFO Queue. In addition, we obtain a model for the connection teardown phase and prove several properties that represent its correctness.

Contents

Acknowledgements

First and foremost, I would like to thank Wan Fokkink for supervising this thesis and providing me with insightful remarks and ideas on how to approach the matter. Furthermore, second supervisor David Williams was always prepared to help out and listen if issues with any of the tools or techniques surfaced.

The weekly meetings of the Theoretical Computer Science Reading Group were a welcome break from the rather solitary effort of writing this thesis. I would like to thank the regular attendees Wan Fokkink, Daniel Gebler, Alban Ponse, Stefan Vijzelaar and David Williams for their interesting talks on varying matters related to formal methods and software verification.

Kees Verstoep provided useful support in a timely manner on the tools used to generate state spaces on the DAS-4 cluster and came with some useful tips that significantly sped up the process.

Carsten Rütz, whom I met during the first week of studying Computer Science and stayed with me during the entire duration of my study, was kind enough to listen whenever I had something to discuss.

Finally and most importantly, I owe a great deal of gratitude to Carolien, who supported me throughout the process and was always there to comfort me.

Chapter 1

Introduction

The Transmission Control Protocol (TCP) plays an important role in the internet, providing reliable transport of messages through a possibly faulty medium to many of its applications. However, of the original protocol only a specification in natural language is available (RFC 793, [36]). When TCP became implemented on a larger scale, several issues and ambiguities surfaced. To clarify these ambiguities and identify and address these issues, a supplemental specification was written (RFC 1122, [8]). This specification, in turn, refers to other documents for detailed descriptions of the solutions. Even with these corrections and ambiguities taken into account, in 1999 a list of known implementation problems was compiled that spans more than sixty pages [32].

The networks that TCP operates over have also changed remarkably over the years, and have now reached a bandwidth and speed that had never been envisioned at the time that TCP was originally defined. Many extensions to increase the performance and reliability of TCP over such networks have been proposed. As a result, TCP as we know it today is specified by a combination of a large number of documents, that describe the protocols behaviour mostly in natural language.

For complex systems such as TCP, formal specification and verification can be used to manage their complexity. The approach is twofold: first, the system is modelled using formal techniques that have mathematical underpinnings, to obtain a specification. To this specification, verification techniques are then applied to detect errors in the system.

Two methods are generally used to verify a formal specification. The first method, generally referred to as *model checking*, defines properties and automatically checks whether or not the system satisfies these properties. Properties can be defined in some modal logic such as Linear Temporal Logic (LTL) or the μ-calculus. In general, we distinguish between safety properties, that state that 'something bad' will never happen, or liveness properties, that state that 'something good' will eventually happen. Alternatively, notions of process equivalence can be used to show that the behaviour of a system is equal to that of another system for which the desired properties are known to hold.

The model checking process can be automated and is relatively quick: properties of a state space can in general be checked in seconds. Hence, the process can be repeated often. Furthermore, an execution trace can be obtained whenever a property is not satisfied. Such a trace is useful to determine whether the property is not satisfied as a result of a modelling mistake or an error in the protocol.

The second method specifies the desired behaviour of the system, again using formal methods, and then verifies whether or not the system's behaviour satisfies this description. This method is generally referred to as *deductive software verification* and involves theorem proving. With this approach, the behaviour of programming constructs such as sequential composition or variable assignment is specified by axioms. By using these axioms, one can construct a proof by applying backwards reasoning, starting from a complex program that consists of a composition of the basic constructs. Generally, the construction of this proof is a laborious process that requires detailed knowledge of both the proof system that is used and the axioms that are applied.

Formal specification and verification techniques may be used for various reasons. First of all, due to

its formal nature, a formal specification serves as an unambiguous reference of the system's behaviour under all circumstances. Second, by using verification to find errors in a system, its reliability can be improved. Finally, the process can be helpful during the development of a system. Over the years, techniques have emerged in which a system's specification is designed by only using formal techniques. The (first) implementation of the system may then be generated automatically from the specification, allowing developers to focus on the intricacies of the system rather than the languages in which they develop their systems.

Problem Statement

This work was initially triggered by a concern that Dr. Barry M. Cook, CTO of 4 Links Limited, relayed in an e-mail about the correctness of one of the extensions that aim to improve the performance and reliability over high-speed, high-bandwidth networks; the Window Scale Option as proposed in RFC 1323 [23]. From his e-mail, we quote: *"The concern I have relates to the Window Scale option for TCP's sliding window [...] The effect is to make the window size reportable only in units of* 2^n *bytes. This may conflict (but may still work) with a requirement that the receive buffer space available (sequence number + window size) should not change downward (or earlier packets may overflow a later size)."*

Our goals for this work are threefold. First of all, we aim to formally specify TCP extended with the Window Scale Option and subsequently verify that that the protocol works correctly. Second, through a literature study that we conducted at the start of our project, of which the results will be discussed in detail in the next section, we found that none of the earlier efforts to formally specify and verify the correctness of TCP considers the Window Scale Option. In addition, earlier verifications of the Sliding Window Protocol, which plays an important role within TCP, also do not take into account this option, nor the fact that the size of the sender's window may be adapted by the receiver to reflect a decrease in the receive buffer space available. Hence, we aim to open up additional features of the Sliding Window Protocol for study by the formal methods community.

Finally, we aim to highlight ambiguities or inconsistencies in the specifications, in order to facilitate future implementers of the protocol. To this end, we will extract our formal specification of TCP directly from the orginal specifications of the protocol and the Window Scale Option, more specifically RFCs 793, 1122 and 1323. This also ensures that our specification conforms to the original RFCs rather than some abstraction of these RFCs devised by others.

To either show that TCP extended with the Window Scale Option is correct, or highlight errors that may occur as a result of the extension, our verification will check whether the extended protocol enables the delivery of a byte stream to the receiver's application layer in the same order as that it was sent out by the sender's application layer. As a basis for this verification, we will first develop a formal specification of the protocol and its extension. We will then apply verification techniques, aided by several toolsets, to prove the correctness of the protocol.

Research Question

Our research will focus on the following question: *"When extended with the Window Scale Option as proposed in RFC 1323, does the Transmission Control Protocol enable the complete and uncorrupted delivery of a byte stream to the receiver's application layer in the same order as that it was sent out by the sender's application layer?"*

Hence, we require that in each run of the protocol extended as proposed in RFC 1323:

1. Every byte that is delivered to the sender's transport layer must be delivered to the receiver's transport layer uncorrupted
2. All bytes are delivered to the application layer of the receiver in the same order as the order in which they were delivered to the transport layer of the sender

The research question gives rise to our hypothesis that including the Window Scale option leads to scenarios in which:

1. There is a byte that is delivered to the receiver's transport layer corrupted, or
2. There are bytes that are delivered to the application layer of the receiver in a different order as the order in which they were delivered to the transport layer of the sender, or
3. There are bytes that are delivered to the application layer of the sender, but never delivered to the application layer of the receiver, or
4. There are scenarios in which the execution of the protocol gets stuck

Our thesis is structured as follows: in the next chapter, we will give a high-level overview of TCP and the Sliding Window Protocol, after which we will discuss the existing work on formal specification and verification of TCP and the SWP (chapter 2). In chapter 3, we will give an overview of the formal techniques that we have used for both our specification and verification. We will then discuss our specification in detail (chapter 4) and document our verification procedure in chapter 5. We end this thesis with our conclusions, and ideas for future work on the specification and verification of TCP (chapter 6).

Chapter 2

TCP

2.1 Introduction

The Transmission Control Protocol (TCP) aims to provide a reliable transport service between two parties that communicate over a possibly faulty network. The protocol is situated in the transport layer of the OSI model [16]. Hence, it receives a byte stream of data from any application and packages this into segments. These segments are then handed to the network layer. This layer is responsible for the forwarding of the segments to the receiving side. On the receiving side, the network layer hands the segments to TCP which is then responsible to ensure that the byte stream is delivered to the receiver's application in the same order as it was sent out by the sender's application.

The responsibilities of TCP can roughly be divided into two categories: connection management and data transmission. As an entity in a network may communicate with many parties at the same time, TCP needs to maintain status information for the byte stream between itself and each of these parties, in order to distinguish the byte streams from each other. This status information is managed using connections; each byte stream is identified by the unique internet address of the remote host combined with a port number that identifies the application that the data is targeted to. Connection management sets up the connections, manages the byte streams and their corresponding states and ensures that connections are closed in a safe manner.

Data transmission involves the transfer of segments from the sender to the receiver. The sender numbers each segment it sends with a sequence number, such that the receiver can determine the correct order of the segments it receives. To ensure that all segments are delivered, the sender periodically retransmits segments until it has received an acknowledgement from the receiver that the segment was delivered correctly. TCP uses the Sliding Window Protocol (SWP) to enable operation of the protocol with only a finite set of sequence numbers. In addition, to control the flow of bytes between sender and receiver, the receiver continuously updates the sender on how much data it is willing to accept.

2.2 The Sliding Window Protocol

One of the first communication protocols that ensured reliable transport of data over a lossy medium was the Alternating Bit Protocol (ABP). In this protocol, a sender sends a segment to the receiver and subsequently waits for an acknowledgement. If such an acknowledgement does not arrive within a specified time-period, the segment is retransmitted. Only after the sender has received an acknowledgement, it is allowed to send the next segment. Consequently, only two sequence numbers are required for correct operation of the protocol. These two sequence numbers can be represented by a single bit that alternates between a value of 0 and 1.

A downside of the alternating bit protocol is that it is inefficient: the sender has to wait for an acknowledgement after each segment it has sent. To alleviate this problem, the Sliding Window Protocol (SWP)

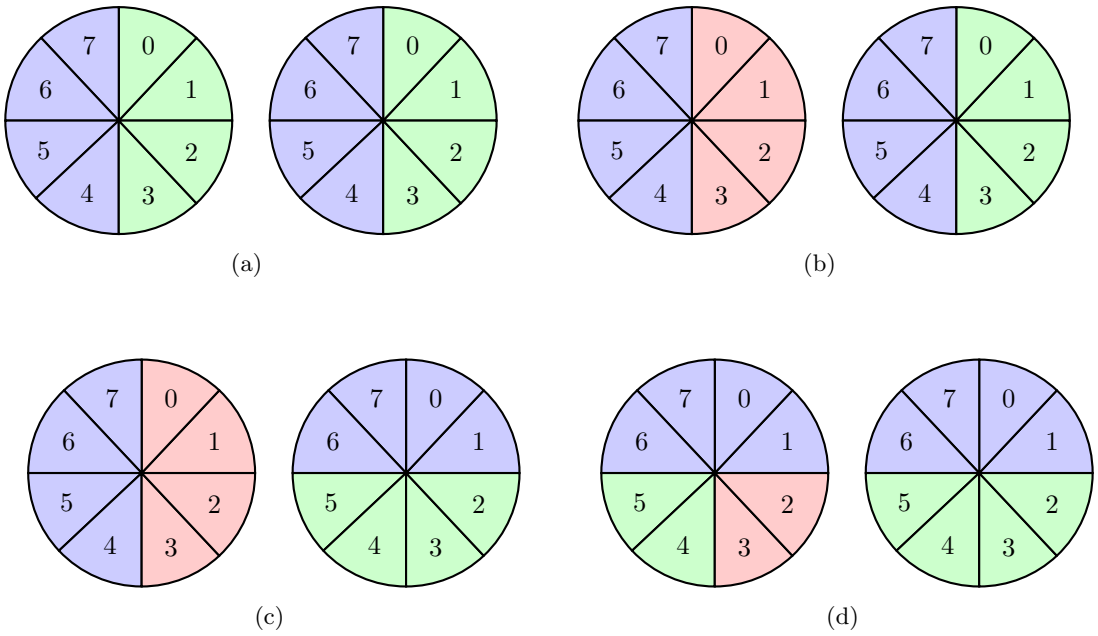


Figure 2.1: Operation of the Sliding Window Protocol

was proposed. In the SWP, n sequence numbers are used ranging from 0 to $n-1$. Both the sender and receiver maintain a window, representing a set of sequence numbers they are allowed to send or receive respectively. The sender may send as many octets as the size of its window before it has to wait for an acknowledgement from the receiver. Once the receiver sends an acknowledgement for m octets, its window 'slides' forward by m sequence numbers. Likewise, the sender's window 'slides' forward by m sequence numbers if this acknowledgement arrives.

For the SWP to function correctly over mediums that may lose data, the maximum size of the window is $\frac{n}{2}$. If window sizes are larger, a retransmission of a segment with sequence number i may be mistaken for a fresh segment with the same sequence number, resulting in a corrupted byte stream. As a consequence, only $\frac{n}{2}$ octet buffers are required at the receiving side. Intuitively, the SWP is a generalisaton of the ABP in which the sequence number space is split into two parts, the first part representing the 0-bit that was used in the ABP and the second part representing the 1-bit.

As an example, consider a sequence number space of size 8 and a window of size 4. Initially, both the sender and receiver have available windows of size 4 as figure 2.1 (a) shows. Now, the sender buffers octets $0 \ldots 3$ and sends two segments, one containing octets 0 and 1 and the other containing octets 2 and 3. The sender will have to wait for an acknowledgement before it can send additional data, as it has used up its entire window space as figure 2.1 (b) shows.

Once the first segment arrives at the receiver, it forwards the octets in the segment to the application layer and sends an acknowledgement. Subsequently, it 'slides' its window by 2, as shown in figure 2.1 (c). The sender also updates its window as it receives this acknowledgement. As figure 2.1 (d) shows, it may now send octets 4 and 5.

The implementation of the SWP that is used in TCP is different from our example, since octets may be acknowledged before they are forwarded to the application layer and therefore still occupy a position in the receive buffer. In this case, the receiving entity reduces the size of the window through the acknowledgement segment that it sends to the sending entity, resulting in a situation as figure 2.2 (a) shows, as opposed to the situation in figure 2.1 (d). By doing this, it ensures that the sending entity does not send new data that will overflow its buffer. Once the octets are forwarded to the application layer, it may reopen the window as figure 2.2 (b) shows. The details of window management in TCP will be discussed in chapter 4.

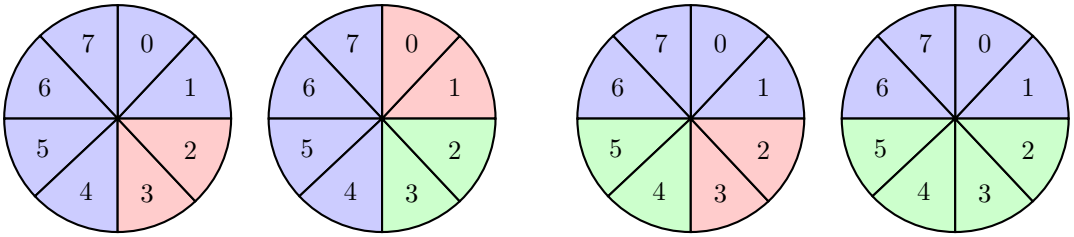


Figure 2.2: Window management in TCP's SWP implementation

2.3 Functional specification of TCP

2.3.1 Segments

Any communication between two TCP entities goes through the exchange of segments over a communication medium. The purpose of a segment is twofold. First of all, a segment may contain zero or more octets of data that an application at the sending process wants to relay to an application at the receiving process. Second, a segment is used to communicate control information between the two entities. To this end, each segment that TCP sends is prepended with a header. Among other things, this header contains the sequence number of the segment, the number of octets that are included in the segment and several flags that influence the behaviour of the protocol.

To prevent segments from lingering around the network forever, a Maximum Segment Lifetime (MSL) is defined. Every time a segment arrives at a hop in the network, the hop verifies whether the 'age' of that segment is smaller than the MSL added to the time that the segment was sent. If this is not the case, the segment is removed from the network by the hop.

For the sake of completeness, we include a detailed discussion of the segment's header below. This section can easily be skipped.

Segment header layout

The header of a segment contains the following fields:

- `SEG.SRC` - **Source port** *(16 bits)*
  The port number of the service that the TCP entity that sent this message is running on.
- `SEG.DST` - **Destination port** *(16 bits)*
  The port number of the service that the TCP entity that this message is sent to is running on.
- `SEG.SEQ` - **Sequence number** *(32 bits)*
  If the `SYN`- or `FIN`-flag is not set, this field contains the sequence number of the first octet in this segment. If the `SYN`-flag is set, this field contains the initial sequence number (`ISN`), and the sequence number of the first octet in this segment is calculated by taking `ISN`+1. If the `FIN`-flag is set, this field contains the sequence number of the `FIN`-segment which is the sequence number of the last octet that the TCP instance sent incremented by 1.
- `SEG.ACK` - **Acknowledgement number** *(32 bits)*
  The next sequence number that the sender of this segment expects to receive. This field is only to be interpreted if the `ACK`-flag is set.
- `SEG.OFF` - **Data offset** *(4 bits)*
  The size of the TCP header as a multiple of 32, to indicate where the data that is included in this segment begins.

- **Reserved** *(6 bits)*
  Empty header space, reserved for future use.
- **Control bits** *(6 bits)*
  - `SEG.URG` - Urgent flag
    Indicates that the urgent function is triggered at the sender of the segment.
  - `SEG.ACK` - Acknowledgement flag
    Indicates that the segment contains acknowledgement information.
  - `SEG.PSH` - Push flag
    Indicates that the push function is triggered at the sender of the segment.
  - `SEG.RST` - Reset flag
    Indicates that the reset function is triggered at the sender of the segment.
  - `SEG.SYN` - Synchronise flag
    Indicates that both entities are synchronising on an initial sequence number.
  - `SEG.FIN` - Finalise flag
    Indicates that no more data will come from the sender and that it wishes to close the connection.
- `SEG.WND` - **Window size** *(16 bits)*
  The number of octets that the sender of this segment is willing to accept.
- `SEG.CHK` - **Checksum** *(16 bits)*
  A checksum calculated over the header and data in this segment by the sender to facilitate integrity checking by the receiver.
- `SEG.UP` - **Urgent pointer** *(16 bits)*
  The sequence number of the last octet of the data that is marked as urgent. Only to be interpreted if the Urgent flag is set.
- **Options** *(variable size)*
  To facilitate enhancements to TCP without breaking the core specification, options may be appended to the end of the header. An option is always a multiple of 8 bits in length.
- **Padding** *(variable size)*
  After the options, padding is included in the header to ensure that it ends on a 32-bit boundary.

2.3.2 Connection management

We begin our functional specification of TCP with a discussion of connection management. At any point in time, an entity may be engaged in communications with several parties. For example, the user of the system that the protocol is running on may be sending an e-mail, while at the same time having a Skype conversation with a friend. In this specific case, it is clear that the network layer has to deal with data from two different applications. The outgoing e-mail data must be directed to the server that is responsible for e-mail handling, while the outgoing data for the Skype conversation should go to the Skype servers. To make matters even more complex, in both cases data must flow from the application layer into the network as well as from the network back to the application layer. Clearly, a mechanism is required to enable TCP to distinguish between the byte streams it exchanges with the remote entities that it communicates with.

This mechanism, called connection management, identifies a connection as a two-tuple of sockets. A socket is itself a two-tuple, containing the unique internet address of a host combined with the port number that identifies the application that the data is targeted to or originating from. Seen from a TCP entity, each byte stream that it handles is bound to a local socket, consisting of the internet address of the local host and the application that this host uses to handle the data, and a remote socket, consisting of the internet address of the remote host and the port number of the application that the remote host

uses to handle the data. Together, this pair of sockets forms a connection. In TCP, each connection is bidirectional, meaning that data may flow in both directions.

This connection can now be used to maintain the state of the communication with any remote entity. For this purpose, the Transmission Control Block (TCB) is introduced. A TCB is kept for every connection and, apart from the local and remote socket numbers, contains variables regarding both outgoing and incoming data.

The following variables are maintained regarding the outgoing data:

1. A pointer to the send buffer; a buffer containing octets that have been accepted as a result of a `SEND` call from the application layer.
2. A pointer to the retransmission queue, on which segments that have been sent are placed until they are acknowledged.
3. `SND.UNA` - The sequence number of the first octet that is sent but not yet acknowledged.
4. `SND.NXT` - The sequence number of the first octet that will be sent next (and may be buffered).
5. `SND.WND` - The number of octets that are allowed for transmission.
6. `SND.UP` - The sequence number of the first octet following the data marked as urgent.
7. `SND.WL1` - The sequence number of the segment used for the last window update.
8. `SND.WL2` - The acknowledgement number of the segment used for the last window update.
9. `ISS` - The initial send sequence number. This is the sequence number of the first segment that the entity will send.

In addition, the following variables are maintained regarding the incoming data:

1. A pointer to the receive buffer; a buffer in which octets that are accepted from the network layer are stored before being forwarded to the application layer.
2. `RCV.NXT` - The sequence number of the next segment that the receiver expects to receive.
3. `RCV.WND` - The maximum number of octets that the entity is prepared to accept at once.
4. `RCV.UP` - The sequence number of the first octet following the data marked as urgent.
5. `IRS` - The initial receive sequence number. This is the sequence number of the first segment that the entity expects and will therefore accept.

Connection establishment

During its lifetime, a connection progresses through several states. Of course, both initially and after finishing the communication the connection does not exist. In this case, the connection's state is described as `CLOSED`. To initiate communications, a connection must be established. Connection establishment is initiated by issuing the `OPEN` call from the application layer to TCP.

Opening a connection can be done either actively, meaning that the initiating side has the intention to send data, or passively, indicating that the initiating side is willing to accept incoming connection requests. Due to this asymmetry, we identify the following two scenarios:

1. The local TCP entity actively opens a connection In this case, the TCP entity instantiates the TCB in which all state information relevant for the connection will be stored. It then sends a segment with the `SYN` flag set and progresses the connection's state from `CLOSED` to `SYN SENT`. While in this state, the TCP entity waits for an acknowledgement of the segment that has the `SYN` flag set. Upon receiving this acknowledgement, it will send back an acknowledgement and progress to the `ESTABLISHED` state.

A variation to this scenario is possible where the local TCP entity receives a segment with only the `SYN` flag set while in the `SYN SENT` state, as a result of the remote entity also actively opening the connection. In this case, the local TCP entity will send a segment with both the `SYN` and `ACK` flags set and progress to the `SYN RCVD` state.

2. The local TCP entity passively opens a connection In the second case, the TCP entity again instantiates the TCB after which the connection's state will progress to `LISTEN`. While in this state, the TCP entity waits until it receives a segment with the `SYN` flag set.[1] Upon receiving such a segment, it responds by sending a segment with the `SYN` and `ACK` flag set and progresses to the state `SYN RCVD`. It will then wait for an acknowledgement of this segment and progress to the `ESTABLISHED` state.

Again, there is a variation to this scenario where a passively opened connection may be transformed into an actively opened connection as the result of a `SEND` call issued from the application layer of the local TCP entity while this entity is in the `LISTEN` state. In this case, the local TCP entity will send a segment with the `SYN` flag set and progress to the `SYN SENT` state, after which the procedure will be as described before.

We see that in both scenarios, first the initiating side sends a `SYN` segment. As a response to this segment, a `SYN, ACK` segment is sent. As this segment is acknowledged, the connection's state at either end progresses to `ESTABLISHED`. This procedure is called the *three-way handshake*, after the three segments that are exchanged during its course.

Once the connection has reached the `ESTABLISHED` state, both entities have reached an agreement on the configuration to use for the connection and either end will have stored the relevant data in their TCB. For both entities, among other things, this data involves the initial sequence number that the entity will use for outgoing data (`ISS`), the size of the send window for the entity (`SND.WND`), indicating the maximum number of octets that it may send at once, and the size of the receive window for the entity (`RCV.WND`), indicating the maximum number of octets that it is prepared to accept at once. From this point on, data transfer between the entities may take place through consecutive calls of the `SEND` and/or `RECEIVE` function from the application layer.

Connection teardown

Unfortunately, nothing lasts forever and therefore, a mechanism is also in place to tear down a connection. If an application at an entity has no more data to send, it issues the `CLOSE` call from the application layer. It is important to note that as a result of issuing this call, the data transfer flowing from the TCP entity that issued the `CLOSE` call to the remote TCP entity will be terminated, essentially transforming the bidirectional connection into a unidirectional one. Only after a `CLOSE` call has been issued from the application layer of the remote TCP entity as well, the connection will be torn down completely. Hence, there are two scenarios that need to be discussed:

1. While in the ESTABLISHED state, the local TCP entity receives a CLOSE call from the application layer Upon receiving a `CLOSE` call from the application layer, the TCP entity will delay the processing of this call until any byte it has buffered in its send buffer is segmentised and sent to the entity at the other end. Then, it will send a segment with the `FIN` flag set, after which the connection will progress from the `ESTABLISHED` state to the `FINWAIT-1` state. While in this state, the TCP entity will no longer accept any `SEND` calls from the application layer, and wait for an acknowledgement of the `FIN` segment to arrive. Once an acknowledgement of the `FIN` segment is received, the connection will progress to the `FINWAIT-2` state.

[1] Note that this segment indicates an actively opened connection from a remote entity.

The connection will remain in the FINWAIT-2 state until the TCP entity receives a FIN segment from the other end, indicating that at the other end of the connection, a CLOSE call was issued from the Application Layer. At this point, the TCP entity that received the FIN will send an acknowledgement and progress to the TIME WAIT state. This state adds a delay of twice the maximum segment lifetime (MSL) before the connection is definitely closed, to ensure that the acknowledgement arrives at the other side. Finally, the connection progresses to the CLOSED state, meaning that all state information for the connection is deleted from the TCP entity.

There is a slight variation to this scenario where the TCP entity receives a segment with the FIN flag set while in the FIN WAIT-1 state, indicating that at the remote entity, a CLOSE call was issued from the application layer as well. In this case, the TCP entity sends an acknowledgement and progresses its state to CLOSING. Upon receiving an acknowledgement of its own FIN segment, the entity will progress to the TIME WAIT state.

2. While in the ESTABLISHED state, the local TCP entity receives a FIN from the network Upon receiving a segment with the FIN flag set, the TCP entity will send an acknowledgement and progress to the CLOSE WAIT state. In this state, the TCP entity will no longer accept RECEIVE calls from the Application Layer but may still accept SEND calls. The TCP entity remains in this state until a CLOSE call is issued from the Application Layer. Upon receiving this call, the TCP entity will send a segment with the FIN flag set and progress to the LAST-ACK state, waiting for an acknowledgement of the segment it just sent. Once this acknowledgement is received, the connection progresses to the CLOSED state.

The closing of the connection marks the end of our discussion of connection management. An overview of the states that we have discussed and the transitions between them is shown by figure 2.3, that we adapted from [8, 36]. Please note that there are some variations to the connection setup scenarios that are not included in our discussion. For a full overview of TCP's connection setup procedure, we refer to [8, 36].

2.3.3 Data transfer

Once the connection has been set up, the data transfer phase may begin. To simplify our discussion, we will distinguish between a sender and a receiver that engage in a unidirectional transfer of data. We assume that the connection is in the ESTABLISHED state. In a real-world scenario, data would flow in both directions, requiring the sender to also act as a receiver and vice versa.

Sender sends data

Data transfer starts at the application layer of the sender, where octets of data that are to be sent to the remote entity can be passed to TCP by (consecutive) SEND calls. The sender maintains a buffer of these octets, the send buffer, which operates as a FIFO queue. As long as there is capacity left in the send buffer, TCP will accept SEND calls from the application layer and put the octets that are passed as arguments to these SEND calls in the buffer.

TCP may send the octets in its buffer at its own convenience. After a single SEND call, it could for example wait for more SEND calls from the application layer before sending out any data. As this behaviour may lead to undesirable delays, two mechanisms are available for the application layer to indicate that the data that it wants to send is important.

First of all, the application layer may set the PUSH flag for the data. If this flag is set, the data must be transmitted immediately to the receiver. Second, the application layer may set the URGENT flag. The purpose of setting this flag is to ensure that the data is processed as soon as possible once it has been put into the receive buffer of the receiver.

TCP uses the sliding window protocol for its data transfer. In this protocol, both the sender and the receiver maintain a window that represents the number of segments that it is allowed to send or willing

to receive respectively. The initial size of the window is agreed upon during connection establishment, such that initially, SND.WND = RCV.WND. The value that is chosen represents the space available in the receive buffer.

To manage its window, the sender maintains the following variables in the TCB: SND.UNA, SND.NXT and SND.WND. At any time, SND.UNA holds the sequence number of the first segment in the sequence number space that was sent, but for which an acknowledgement has not yet been received. SND.NXT holds the sequence number of the next segment that the sender will send. Finally, SND.WND holds the number of octets that TCP may send at most before it should wait for an acknowledgement. Initially, both SND.UNA and SND.NXT are set to the initial sequence number that both parties have agreed upon during connection establishment. Likewise, SND.WND is set to the size of the window that both parties have agreed upon.

The actual number of octets that TCP can send at a certain point in time is calculated by taking the difference m between SND.UNA and SND.NXT. If $m <$ SND.WND, TCP may package any number of octets $n \leq m$ that it thinks reasonable into a segment and send it into the medium. Subsequently, TCP does several things:

1. The octets that were included in the segment are removed from the send buffer.
2. The segment that was sent is put on the retransmission queue.
3. A retransmission timer is started for the segment.
4. SND.NXT is advanced by n, now indicating the sequence number of the octet that will be sent next.

If the retransmission timer expires before the sender receives an acknowledgement of the segment, the segment will be retransmitted and the timer restarted. By doing this, any segment that is not accepted at the remote end, for whatever reason, is retransmitted until it is eventually accepted exactly once. This process may repeat itself as long as $m <$ SND.WND. By default, TCP uses a go-back-n retransmission scheme. However, the protocol may keep segments that arrive out of order to employ a selective repeat retransmission scheme. This scheme can be optimised even further by implementing the selective acknowledgement extension [13].

Some ambiguity surrounds the specification of the sequence number, as both octets and segments are assigned one. It is important to note that in principle, TCP numbers each octet with a unique sequence number, modulo the size of the sequence number space. A segment inherits its sequence number from the first octet that it contains. However, if a segment does not contain any octets, it still requires a sequence number. In this case, the sender will still number the segment with the sequence number as maintained in SND.NXT, but SND.NXT will not be updated.

As an example, suppose that we have a sequence number space of 4 sequence numbers. The following scenario may occur. First, TCP sends out a segment that contains 2 octets. This segment will carry the octets numbered 0 and 1 and have sequence number 0. Shortly afterwards, TCP will send a segment without any octets to relay some control information to the other entity. Since SND.NXT is now set to 2, this segment will have sequence number 2. After this segment, TCP again sends out a segment that contains 2 octets. SND.NXT is still set to 2 and consequently, this last segment will also have sequence number 2 and include octets 2 and 3.

SYN and FIN segments, which are used during connection setup and teardown, form an exception to this rule, as is stated on page 26 of [36]: *"The* SYN *segment is considered to occur before the first actual data octet of the segment in which it occurs [if any], while the* FIN *is considered to occur after the last actual data octet in a segment in which it occurs."* Hence, if a SYN segment is sent with sequence number n, this same sequence number must not be used to send a data segment after this SYN segment until the sequence number space wraps. Likewise, if the last byte that is sent on a connection has sequence number n, the FIN segment that is subsequently used to close the connection gets sequence number $n + 1$. In both cases, SND.NXT is updated accordingly after sending the control segment.

A segment arrives at the receiver

If all goes well, after the transfer through the medium a segment will eventually arrive at the receiver. Recall that in its TCB, the receiver maintains a pointer to the receive buffer and several variables. Of importance here are the variables RCV.NXT and RCV.WND. Initially, RCV.NXT is set to the initial sequence number that the sender has communicated to the receiver during connection setup and RCV.WND is set to the capacity of the receive buffer.

As a first check on the segment that arrived, the receiver will verify that the segment is acceptable. In [36], a segment is defined as acceptable in the following situations:

1. If the segment does not contain data octets:
   (a) If RCV.WND $= 0$, it is required that SEG.SEQ $=$ RCV.NXT
   (b) If RCV.WND > 0, it is required that RCV.NXT $\leq$ SEG.SEQ $<$ (RCV.NXT + RCV.WND)
2. If the segment does contain data octets
   (a) If RCV.WND $= 0$, the segment is not acceptable.
   (b) If RCV.WND > 0, it is required that
      - Either RCV.NXT $\leq$ SEG.SEQ $<$ (RCV.NXT + RCV.WND)
      - Or: RCV.NXT $\leq$ SEG.SEQ + SEG.LEN$-1 <$ (RCV.NXT + RCV.WND)

If the segment is acceptable, processing continues as specified below. Otherwise, the segment is dropped and an acknowledgement is sent to the sender containing the current value of RCV.NXT.

After the initial acceptability check, segments are processed in order of their sequence numbers. Segments that arrive out of order may be dropped by the receiver. However, to improve performance, the specification suggests that these segments are held in a special buffer to be processed as soon as their turn arrives.

The second step in the processing of a segment is to check whether any of the following control flags is set: RST, SYN, ACK and URG. Of these flags, the RST and SYN flags can only be set during connection setup or as a consequence of errors during connection setup. Therefore, we will not discuss the behaviour of the protocol in response to such a flag being set here.

Because of the fact that we discuss a unidirectional scenario here, the segment will not contain (new) acknowledgement information. Hence, the behaviour of the protocol in response to the ACK flag being set will also not be discussed here but is delayed until the following section, where the sender receives an acknowledgement of a segment it sent earlier.

If the URG flag is set, the variable RCV.UP as maintained in the TCB is set to max(RCV.UP,SEG.UP). Furthermore, if RCV.UP $>$ SEG.SEQ + SEG.LENGTH, the application layer is signalled that the remote entity has urgent data.

Following the processing of these flags, the actual octets that are included in the segment may be processed. First of all, the octets are taken from the segment and written into a buffer at the receiver. Then, it is checked whether the PUSH flag was set. If this is the case, the application layer is informed that there is data in the receive buffer that the sender marked with the PUSH flag. Finally, RCV.NXT is advanced by the number of octets that have been accepted.

The receiver must acknowledge the fact that it took responsibility for the data in the segment to the sender, and to this end, an acknowledgement containing the new value of RCV.NXT - reflecting the next sequence number that the receiver expects to receive - is constructed and sent back to the receiver.[2] This sets the SWP implementation of TCP apart from other implementations, as an acknowledgement is sent while the octets may not yet be forwarded to the application layer and therefore occupy a position in the receive buffer. Therefore, the size of the window that is reported back to the sender represents

[2]When the data transfer is bidirectional, this acknowledgement may be piggybacked onto an outgoing segment carrying data under the condition that this does not incur an inappropriate amount of delay.

the available capacity in the receive buffer, if this capacity is less than the difference between RCV.NXT and the size of the receive window that was originally agreed upon permits. In naive implementations, each acknowledgement will carry the updated size of the receiver's window. Since this may degrade performance, several strategies have been proposed to minimise the negative effects window management may have. These strategies, however, are outside the scope of our project and will therefore not be discussed here.

Finally, the receiver will verify whether the FIN flag was set in the incoming segment. If this was the case, the receiver will progress from the ESTABLISHED to the CLOSE-WAIT state.

An acknowledgement arrives at the sender

Once an acknowledgement arrives at the sender, the sender verifies whether SND.UNA $<$ SEG.ACK $\leq$ SND.NXT. If this is the case, SND.UNA is set to SEG.ACK and all segments on the retransmission queue that contain octets with sequence numbers $n \ldots m <$ SEG.ACK are removed. Furthermore, if SND.UNA $\leq$ SEG.ACK $\leq$ SND.NXT $\wedge$ (SND.WL1 $<$ SEG.SEQ $\vee$ (SND.WL1 $=$ SEG.SEQ $\wedge$ SND.WL2 $\leq$ SEG.ACK)), the send window must be updated. This is done by setting SND.WND to SEG.WND, SND.WL1 to SEG.SEQ and SND.WL2 to SEG.ACK.

If it is not the case that SND.UNA $<$ SEG.ACK $\leq$ SND.NXT, there are two possibilities. In case SEG.ACK $\leq$ SND.UNA, the acknowledgement can be ignored since it is a duplicate. If SEG.ACK $>$ SND.NXT, the sender will send an acknowledgement, drop the segment and return. This last situation can only occur in case of a bidirectional data transfer.

2.4 Known problems

Several issues with TCP's implementation have surfaced as the networks that the protocol operates over have become more sophisticated. In this section, we will discuss the problems that are relevant to our project.

2.4.1 Sequence number reuse

The first problem that we will discuss is that of sequence number reuse. Since the sequence number field in the TCP header only allows for a finite 32-bit sequence number, the protocol can only use sequence numbers up until $2^{32} - 1$ to number the octets that it sends. As a result of this, after 2^{32} octets have been sent the next octet will again have a sequence number equal to the initial sequence number.

In RFC 793, it is stated that *"the duplicate detection and sequencing algorithm in the TCP protocol relies on the unique binding of segment data to sequence space to the extent that sequence numbers will not cycle through all 2^{32} values before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segment have 'drained' from the internet"* [36].

The 'draining' that the specification refers to here is achieved by enforcing the Maximum Segment Lifetime (MSL) at each hop in the network. Since TCP has a sequence number space of size 2^{32}, this assumption is automatically met for a reasonable MSL of two minutes and networks up to a speed of 17.9 megabytes per second.[3] However, as network speeds increased, scenarios started to become possible where a sender could send its entire sequence number space into the network in an amount of time shorter than the MSL.

The following simplified scenario shows why this is a problem. Suppose that we have a sequence number space of size 8 and a send window of size 2. The sender sends a segment x into the network containing

[3] Namely: sending the 2^{31} bytes in the 'second half' of the sequence number space must take at least two minutes. Hence, the maximum speed is calculated by taking $2^{31}/120$.

octets 0 and 1. This segment arrives at the receiver, which subsequently responds with an acknowledgement. This acknowledgement, however, is delayed in the medium, causing the retransmission timer to go off and segment x to be sent into the medium again. Immediately thereafter, the acknowledgement arrives at the sender and the sender will respond by sending a segment y carrying octets 2 and 3.

Now, let segment y overtake segment x and arrive at the receiver first. Since the receiver expects octet 2, it will accept the segment, buffer the octets it contains and send an acknowledgement. This is repeated with segment y' – carrying octets 3 and 4 – and segment y'' – carrying octets 5 and 6.

Note that, as the entire sequence number space is consumed, `RCV.NXT` is set to 0 again after the receiver has accepted segment y''. Hence, when the acknowledgement of this segment arrives at the sender, the sender will respond by sending a segment z carrying octets 0 and 1.

Meanwhile, segment x arrives at the receiver. The receiver cannot distinguish segment x from segment z since both have sequence number 0 and contain octets 0 and 1. Therefore, the receiver will accept segment x and send an acknowledgement for it. Consequently, once segment z arrives, it will be dropped.

It is immediately clear that this scenario may easily lead to data corruption that goes by unnoticed. To resolve this problem, RFC 1323 ([23]) proposes a mechanism called PAWS, an acronym for Protection Against Wrapped Sequence Numbers.

If this mechanism is implemented, to every segment a timestamp `SEG.TSval` is added that is monotone non-decreasing in time. Furthermore, the receiver maintains the additional variables `TSrecent` and `Last.ACK.sent` in its TCB. Whenever an acknowledgement segment is sent, `Last.ACK.sent` is set to the value of `SEG.ACK`.

If a segment arrives at the receiver for which `SEG.TSval` $<$ `TSrecent`, the segment is dropped and an acknowledgement is sent to the sender containing the current value of `RCV.NXT`. If `SEG.TSval` $\geq$ `TSrecent`, it is checked whether `SEG.SEQ` $\leq$ `Last.ACK.sent`. If this is the case, `SEG.TSval` is stored in `TSrecent`. Regardless of the outcome of this test, processing continues as specified in RFC 793 ([36]).

Essentially, by implementing PAWS the sequence number of a segment is transformed from a single value into a two-tuple. It is important to note here that these timestamps are themselves 32-bit unsigned integers in a modular 32-bit space (again due to the restrictions of the TCP header). Hence, the problem is only moved forward.

It is crucial to understand that there is no other protection against wrapped sequence numbers than the assumption that whenever a connection enters a fragment of the sequence number space for the $n+1$th time, all segments that were sent into the network while the connection was in the same fragment of the sequence number space for the nth time have drained from the network due to the expiry of their MSL. By choosing the values for the timestamp clock wisely, the implementation can be stretched to cover any value for the MSL that is still reasonable.

2.4.2 Performance loss due to small window size

While the second problem that surfaced when networks became more sophisticated does not break the correctness of TCP, it does have a negative influence on the performance of the protocol. Before we discuss the problem in detail, we must first emphasise that there is a direct relation between TCP's performance and the size of the send window. The larger this window is, the more data a sender can send without having to wait for an acknowledgement. It is then also easy to see that in an optimal scenario, the size of the window allows the sender to send as much data into the medium as it can hold at most.[4]

However, the receiver may adjust the size of the sender's window at any time, through the value of `SEG.WND` set in acknowledgement segments that are transferred from receiver to sender. By the fact that the size of this field is limited to 16 bits, the maximum size of the send window is 2^{16}. This means that TCP can send at most 2^{16} octets into the medium before having to wait for an acknowledgement.

[4]This, in turn, is defined by taking the product of the transfer rate and the round-trip delay. For more details, we refer to RFC 1323 ([23]).

Hence, if the medium can hold more than 2^{16} octets, unnecessary delay will be introduced into the communication due to the restriction on the window size.

To resolve this issue, RFC 1323 proposes the Window Scale Option. If this option is implemented, the variables `SND.WND` and `RCV.WND` are maintained as 32-bit numbers in the TCB of the sender and receiver respectively (instead of as 16-bit numbers). Furthermore, the TCBs of the sender and receiver are extended with a variable `SND.WND.SCALE` and `RCV.WND.SCALE` respectively.

During connection setup, the sender and receiver agree upon the value for these variables. This process is straightforward: a TCP entity that actively opens a connection sets `RCV.WND.SCALE` and includes this value in the Window Scale Option header field of its synchronise segment. By doing this, it indicates which scale factor it will apply to the window field of each outgoing acknowledgement if window scaling is enabled for the connection.

When the remote entity receives the synchronise segment, it stores the window scale in the variable `SND.WND.SCALE` and sends an acknowledgement. If this acknowledgement also contains a scale factor, window scaling will be enabled for the connection. From then on, the scale factors to use during the lifetime of the connection are fixed. Since TCP is a bidirectional protocol, a scale factor is agreed on for each direction. It is important to note that these factors may differ from each other.

During the data transfer phase, whenever an entity receives an acknowledgement, it left-shifts the value of `SEG.WND` by the value of `SND.WND.SCALE` before it updates its send window. Likewise, whenever an entity sends an acknowledgement it sets the window field of the outgoing segment to the size of its receive window (`RCV.WND`), right-shifted by the scale factor `RCV.WND.SCALE`.

Implementing the Window Scale Option theoretically enables a window size that is equal to the size of the sequence number space. Therefore, it introduces an additional problem with sequence number reuse that did not occur previously. According to [44], the sliding window protocol functions correctly for window sizes up until 2^{n-1} given a sequence number space of size 2^n, assuming that the medium does not support reordering. However, in the environment in which TCP is used, this assumption does not hold. In [9], a scenario is given where a segment s_0 ranging over the first half of the sequence number space is erroneously accepted twice as a result of duplicating s_0 and a subsequent reordering with a segment s_1 ranging over the second half of the sequence number space.

Another example of the problem that may occur as a result of the fact that the assumption as given in [44] does not hold in TCP's context is shown by the following scenario, in which we have a sequence number space of size 2^3 and a window of size 2^2. The sender starts by sending a segment x containing octets $0 \ldots 3$. After receiving an acknowledgement of this segment, the sender responds by sending a segment x' containing octets $4 \ldots 7$. Again, this segment is acknowledged, after which the sender will send a segment y containing octets $0 \ldots 3$. At this point, the receiver's window ranges over octets $0 \ldots 3$. If segment y arrives, the receiver will accept it, update the window to range over octets $4 \ldots 7$ and send an acknowledgement. Before this acknowledgement arrives at the sender, segment y is retransmitted. Immediately thereafter, the sender receives the acknowledgement, updates its window and sends a segment z containing octets $4 \ldots 7$. Now, let segment z overtake the retransmitted segment y in the medium and arrive at the receiver. The receiver will accept the segment, update its window range to $0 \ldots 3$ and send an acknowledgement. Shortly thereafter, the retransmitted segment y arrives. This segment is now accepted as a regular, in-sequence segment resulting in a corrupted byte stream.

It is crucial to understand that enforcing the assumption on the MSL does not help us here since segment z is sent shortly after segment y was retransmitted. Therefore, this scenario is completely reasonable. To fix this issue, RFC 1323 enforces that the window size is at most 2^{n-2} with n the number of bits available for the sequence number. Now, when the sender retransmits a segment y carrying octets $6 \ldots 7$ and shortly thereafter receives an acknowledgement for this segment, it will respond by sending a segment z carrying octets $0 \ldots 1$. If z overtakes y and arrives at the receiver, its window will be updated to range over octets $2 \ldots 3$. As a result of this, segment y will not be accepted if it were to arrive. Combined with the assumption that segment y will have drained from the network by the time that the receiver's window ranges over $0 \ldots 1$ again, correctness is preserved.

2.5 Related work

In the literature, we distinguish two approaches to the verification of TCP, that may be used alongside each other. The first approach involves a protocol specification P (specifying the protocol itself), a service specification S (specifying the service provided by the protocol to its users) and an underlying service specification U (specifying the characteristics of the medium over which the protocol is going to operate). A verification compares the service provided by P operating over U with S, to see that the protocol exhibits the desired external behaviour.

The second approach also involves a formal specification of both a protocol and the medium that it operates over, but instead of stating correctness in terms of desired behaviour, safety and liveness properties are defined. A verification then checks a model that is generated from the specifications, to ensure that the safety and liveness properties hold in all of its states.

Several attempts have been undertaken to verify (parts of) TCP. These attempts either focus solely on the sliding window protocol, or on TCP in general. In the second case, verifications are typically performed at a higher level; from the establishment to the closing of a connection. Furthermore, multiple incarnations of a connection may be considered.

2.5.1 Specifications and verifications of the Sliding Window Protocol

The Sliding Window Protocol (SWP) aims to ensure reliable, in-order delivery of messages using only a fixed, finite set of sequence numbers and therefore plays an important role within TCP. Table 2.1 shows an overview of studies into the correctness of this protocol that we have studied. Madelaine & Vergamini [27] have modelled the protocol in the specification language `LOTOS` and verified their model using the verification tool `AUTO`. They consider the unidirectional case, in which a sender sends messages over a faulty medium of capacity one to a receiver, which in turn communicates acknowledgements back to the sender over another faulty medium. Their faulty media are specified such that they may lose, duplicate and reorder messages. In their verification, they consider a window size of two, and show that the global behaviour of the protocol is that of a four-place buffer that transmits messages in the correct order, with no loss nor duplication. From this equality, they conclude that the protocol recovers from faults in the media. Of interest is the fact that they recognise that this approach of showing equivalence of the global behaviour to another, known protocol is more difficult if the model is bigger, or even impossible if there is no predefined specification that the protocol can be proven equivalent to. Therefore, they also prove two partial properties – (1) the inputs/outputs are correctly sequenced and (2) an input is always followed by the corresponding output – to show that even in these cases, some form of verification, be it partially, can be undertaken.

Author(s)	**Window size**	**Medium capacity**	**Lose messages**	**Duplicate messages**	**Reorder messages**	**Message direction**
Madelaine & Vergamini [27]	2	2	√	√	√	⇒
Van de Snepscheut [45]	n	1	√	√		⇒
Bezem & Groote [2]	1	1	√			⇔
Fokkink et al. [15, 1]	n	1	√			⇒ [15] ⇔ [1]
Chkliaev et al. [9]	n	n	√	√	√	⇒

Table 2.1: Comparison of the Sliding Window Protocol verifications

In [45], van de Snepscheut also considers the unidirectional case with faulty media of capacity one that may only lose or duplicate data. At first, the protocol is specified as a sequential program that numbers messages from $0 \ldots n$. It is then proved that in this program, the sequence of messages received by the receiver is a prefix of the sequence of messages sent by the sender, and that the difference in length is at most window size n. Furthermore, it is shown that in each state, the program will eventually progress to another state. Then, this sequential program is refined to use bounded sequence numbers, after which it is partitioned into separate processes; a sender, a receiver and two faulty media. Each of these transformations preserves the correctness of the program. Finally, it is shown that if this program

is specialised to a window size of one, the alternating bit protocol is obtained. According to the author, this protocol is verified extensively in the literature. Of particular interest in this paper is that the author states that verifications do not need to consider the case in which messages are garbled since this is equivalent to loosing a message, given that a mechanism is in place to detect that messages are garbled.

Bezem & Groote [2] take an approach similar to that of Madelaine & Vergamini, showing that the SWP with a window size of one has the same external behaviour as a bidirectional buffer of size two. They consider the bidirectional case in which messages are transmitted over unreliable media of capacity one that may lose messages. They use μCRL to specify the protocol, the bidirectional buffer and to prove the equivalence. They claim that their effort gives a structure for generating a correctness proof that is more general and different from existing verifications, and that it shows that verification of more complicated systems in process algebra is possible.

Fokkink et al. [15] also use μCRL to specify the SWP. They then verify the correctness of the protocol for arbitrary window size n, by first eliminating modulo arithmetic and then showing that the specification without modulo arithmetic is branching bisimilar to a FIFO queue of size $2n$. In their specification, the communication media may only lose messages and have a capacity of one. This work was later extended [1] to consider the bidirectional case in which acknowledgements are piggybacked onto data that is sent in the other direction.

Chkliaev et al. [9] go even further than just verification by stating that the SWP is unnecessarily complex. To alleviate this complexity, they specify and verify an improved version, in which the sender and receiver no longer have to synchronise on the sequence number to start with. To achieve this, they require the sender to stop and wait for the maximum message lifetime after it has received an acknowledgement for the message with the maximum sequence number. After this waiting period, the sender can be sure that any message that is still in transit has been cleared from the media and thus safely resume the transmission. They formalise their protocol in `PVS` and use its interactive proof checker to prove, for arbitrary window size and media that may lose, reorder or duplicate data, that all states of the protocol are safe, meaning that in each of these states, the output sequence is a prefix of the input sequence. In their paper, they also highlight an important issue that occurs if the original sliding window protocol is used in combination with media that may lose, duplicate and reorder messages, by giving a scenario in which a message is received twice after an occurrence of message duplication followed by reordering. To solve this issue, additional restrictions must be placed on the protocol to recognise sequence numbers correctly.

Apart from the work by Chkliaev et al., which discusses an adapted SWP, none of the studies conforms to the context in which TCP is generally implemented, where media have an arbitrarily high capacity and may lose, reorder or duplicate data. Furthermore, none of these verifications takes the dynamic character of the sliding window implementation in TCP into account – where the window size may vary depending on the buffer capacity of the receiver – nor the fact that in TCP connections are bidirectional.

2.5.2 Specifications and verifications of TCP

Several publications aim to formally specify and verify the correctness of TCP, of which table 2.2 shows an overview. Murphy & Shankar [30] specify a protocol with a service specification that is very similar to the original service specification of TCP as defined in RFC 793. They first define the service specification, consisting of the events at the user interface of their protocol and the allowed sequences of events for each interface. By a method of step-wise refinement, they then define a protocol specification that responds to events occurring in the service specification, while maintaining several correctness properties. This protocol specification is first constructed for a perfect network service, then for a network service that may lose messages in transit and finally for a network service that may lose, reorder and duplicate messages in transit. The need for a three-way handshake (at step 2) and strictly increasing incarnation numbers (at step 3) becomes apparent with the introduction of possible faults in the medium, explicitly showing why these facilities are present in TCP.

Smith [40, 41] uses a similar approach to the problem. In addition to a service specification, he also

Authors	RFC 793	RFC 1122	RFC 1323	Other extensions	Connection management	Data transfer	Lose messages	Duplicate messages	Reorder messages	Service specification	Protocol specification	Message direction	Connection incarnations
Murphy & Shankar [30]	√				√		√	√	√	√		⇔	n
Smith [40, 41], Smith & Ramakrishnan [39]	√			√	√	√	√	√	√	√	√	⇒	n
Schieferdecker [38]	√	√		√	√				√	√	√	⇔	2
Billington & Han [3, 4, 5, 18, 19, 20, 21, 22]	√	√			√	√	√		√	√	√	⇔	1
de Figueiredo & Kristensen [12]	√	√		√		√	√					⇒	1
Bishop et al. [6], Ridge et al. [37]	√	√	√	√	√	√	√	√	√	√	√	⇔	n

Table 2.2: Comparison of earlier verifications of TCP

gives a protocol specification that is based on TCP as defined in RFC 793. By means of a refinement mapping, it is shown that the protocol specification satisfies the specification of the user-visible behaviour. Using a similar approach, Smith & Ramakrishnan [39] model the behaviour of TCP with selective acknowledgements and verify that it ensures reliable, in-order delivery of messages.

Schieferdecker [38] shows that there is an error in TCP's handling of the `ABORT` call, by giving a (unlikely) scenario in which (1) the `RESET` segment that is sent as a result of the `ABORT` call gets lost, (2) a new incarnation of the connection is set up with an initial sequence number that is smaller than the sequence number of data that is still in transit after which (3) old data is mistaken for new data. After stating a possible solution for this problem, the author gives a summary of a specification effort of TCP in `LOTOS`. Then, several correctness requirements (stated in μ-calculus) are verified using `CADP`. However, to prevent the state space from becoming too large for the hardware available at the time of writing, the specification had to be simplified drastically; by completely removing the data transfer phase from it and restricting the faults in the medium to reordering only.

Billington & Han have studied TCP extensively. In their work, they consider both RFC 793 and RFC 1122. They give a service specification, a protocol specification and a transport service specification using Coloured Petri Nets [24]. A concise overview of their TCP service specification is given in [3] and includes connection establishment, normal data transfer, urgent data transfer, graceful connection release and abortion of connections.

Because of the complexity of TCP, their work is separated in work on connection management and work on the data transfer part of TCP. In [19], they give a model of the connection management service that considers the opening and closing of connections. The specification uses non-lossy media that may delay or reorder messages and does not include retransmissions or multiple incarnations of connections. Sequence numbers are specified without modulo arithmetic, as connection management only uses a small range of numbers compared to the size of the total set of sequence numbers that is typically used. In this paper, they verify connection establishment for two cases: the client-server case, where one entity actively initiates a connection and the other entity passively initiates a connection, and the case where both entities simultaneously initiate a connection. They perform a reachability analysis on their model, finding the states without outgoing transitions (dead markings). For these states, they examine the sequences that led to them to see whether these sequence are acceptable and correct considering the service specification. They find that when a connection is initiated by both entities simultaneously, two acknowledgements are sent by each entity before they enter the `ESTABLISHED` state, where one acknowledgement would suffice. This of course causes unnecessary delay in the establishment of connections.

In [21] this model is further refined to include media that may lose, delay and reorder packets and to consider the regular closing and irregular termination of connections. This revised specification is used as a basis for a verification of connection management [22] considering a model without retransmissions (Model 1) and a model with retransmissions (Model 2). Both models again operate over a non-lossy medium that may delay or reorder messages. Apart from connection establishment, this verification also includes connection termination. As a result of this verification, Billington & Han find that a deadlock may occur in Model 1, when one entity opens the connection passively and, after receiving and acknowledging a connection request, immediately closes the connection again. The protocol then hangs with the initiator of the connection in the `ESTABLISHED` state and the other entity in the state `FIN_WAIT_1`. To make matters even worse, if the initiator decides to close the connection, it will be left waiting for the other entity to complete the release, while this entity is waiting for the initiator to send an acknowledgement of its request for connection termination. This deadlock can be alleviated by including retransmissions, as is shown by a verification of Model 2. The authors claim that it is strange that retransmissions of messages are required in the case of a non-lossy medium. In [18], two additional issues are discussed. One issue describes a scenario where the connection may fail to establish when it is opened simultaneously. The other issue describes a scenario where the entities may get stuck when closing a connection before it reaches the `ESTABLISHED` state.

The work by Billington & Han on data transfer has not yet led to a verification. To avoid having a model with an infinite state space, they define a specification of TCP data transfer that can be parameterised by the size of the medium [4, 5], yielding an infinite set of finite state spaces. Furthermore, they have developed a technique to calculate the size of the state space for a given medium size and show that it grows exponentially with the size of the medium. Finally, they show that for any medium size, the state space of their model is a strongly connected graph and that the automaton is deterministic. Hence, they have obtained a deterministic finite state automaton that models TCP's data transfer for arbitrary medium capacity.

de Figueiredo & Kristensen [12] have also used Coloured Petri Nets, to specify TCP combined with the four most commonly implemented congestion avoidance algorithms. Their model only considers unidirectional data transfer and does not include TCP connection management. Simulations of the model are used to investigate the performance of TCP Tahoe (TCP as specified in RFC 793 and RFC 1122, extended with *fast retransmit* [42]) and TCP Reno (TCP Tahoe extended with *fast recovery* [42]) in the event of one, two or three packet losses.

Instead of verifying TCP specifications, Bishop et al. [6] look at execution traces generated from real-world implementations of TCP, to see if they are accepted by a protocol specification of TCP in Higher Order Logic (HOL). Their protocol specification describes the operation of TCP as a result of calls at the application layer or as a result of receiving a message at the network layer, and includes many options such as PAWS, the window scaling option and congestion control algorithms. Of all of the test traces that they generated, their specification accepts 91.7%. Some of the traces are rejected due to issues in the specification, while others cannot be verified due to memory issues. The authors claim that their test traces give reasonable coverage of TCP. In [37], this work is extended by Ridge et al. to also include a service specification.

While some effort has already been put into a formal specification and verification of TCP, most of the papers we found are in some way incompatible with the protocol as we know it today. The service specification of Murphy & Shankar includes an adapted termination of connections for simplification. Additionally, they construct their own protocol specification for which they then verify several safety and progress properties. Smith constructs a service specification and a protocol specification based on TCP and shows that the protocol specification satisfies the service specification, but has adapted several calls at the user interface of TCP. de Figueiredo & Kristensen consider both RFC 793 and RFC 1122 as well as several congestion avoidance algorithms in their specification but only use this specification to analyse the performance of two TCP variants, while Schieferdecker has to simplify her specification greatly to make automated verification feasible.

Bishop et al. have based their specification on real-world implementations of TCP and include many options, but their work does not relate the specification as they have obtained it to the official specifications of TCP, as they consider the implementations to be the *de facto* standard. Furthermore, they do

not provide a formal verification.

The work of Billington & Han is most extensive. Apart from specifying both the connection management and data transfer service of TCP, they have verified the connection management service in various scenarios and found some issues for which they have proposed and verified solutions. However, their work is still incomplete for several reasons. First, it only considers one incarnation of a connection. Second, their specification does not include timing constraints included in the TCP specification. Third, TCP is nowadays often implemented with many options or extensions that are not considered in their work. Finally, they have as of yet not given a verification of their specification of TCP's data transfer service.

Summary

In this chapter, we gave a high-level overview of TCP and the Sliding Window Protocol. We then gave a functional specification of TCP, based on the original protocol specifications and introduced the Window Scale Option. Finally, we discussed earlier work on formal specification and verification of TCP and the Sliding Window Protocol that we found during a literature study. We showed that none of the earlier efforts incoporate the Window Scale Option. Furthermore, none of the studies completely conforms to the context in which TCP is generally implemented[5], as can be seen from tables 2.1 and 2.2.

[5]Except for the work of Bishop et al. [6] and Ridge et al. [37] which does not relate to the original specifications of TCP.

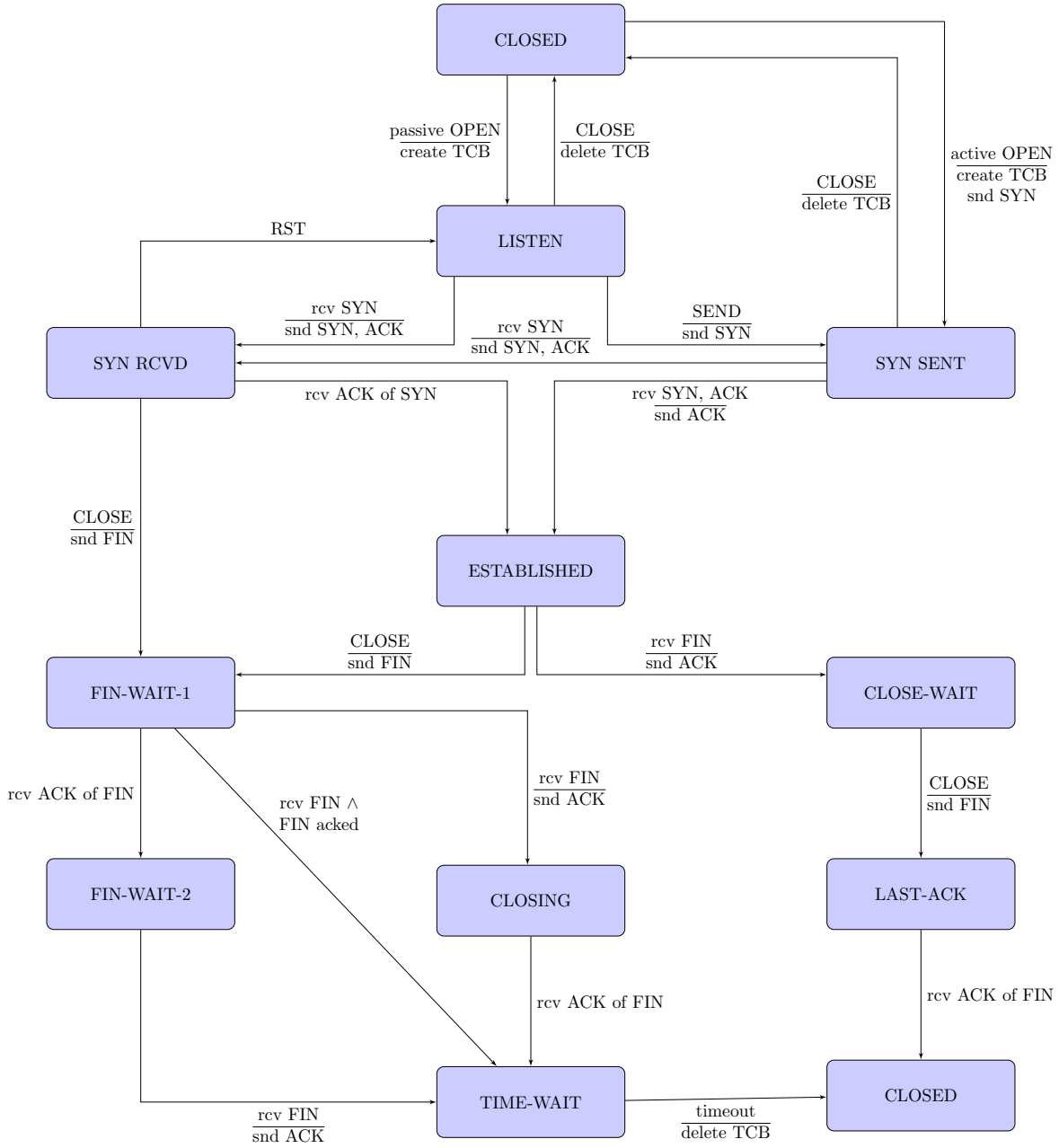


Figure 2.3: The Connection Teardown Procedure

Chapter 3

Process algebra

Process algebras are used to formally specify the behaviour of (concurrent) systems. In general, first the separate components that make up the system are specified. Then, the components are put in parallel, together with a specification of ways for the components to interact with each other. Finally, the initial state of the parallel specification is denoted.

The formal specification of the system's behaviour can then be used to verify the correctness of the system. In chapter 1, we distinguished between deductive software verification and model checking. In this chapter, we will concentrate ourselves on model checking and explain two techniques that we will apply in this thesis: a proof of process equivalence, where (a part of) the behaviour is compared with the behaviour of another system, and property checking, where properties of the system are checked on its state space.

In this section, we will discuss the process algebra μCRL, which distinguishes itself from other process algebras through its ability to handle data. The other notions that we discuss here carry over to other process algebras. For a more complete introduction, we refer to [14]. For a concise overview of formal methods and their applications in software verification, we refer to [33].

3.1 Process specification

A process specification consists of a set of processes. Each process is declared through a *process declaration*, which is in turn built up of process terms.

Process terms

The most atomic form of a process term is an *action*. Actions represent atomic events that occur during the execution of a process. An action may carry zero or more data parameters, indicating the data that is relevant for the execution of the action.

Process terms may be composed to form more intricate behaviour. Two types of composition are generally available; sequential and alternative composition. A *sequential composition* of two process terms t_1 and t_2, denoted $t_1 \cdot t_2$, represents the process that first executes the process as described by term t_1 and, after successful termination, executes the process as described by term t_2.

An *alternative composition* of two process terms t_1 and t_2, denoted $t_1 + t_2$, represents the process that executes either the process as described by term t_1 or the process as described by the term t_2.

Besides alternative composition, a process term can also reflect a deterministic choice based on a condition through composition with the *conditional operator*. A process term of the form $p \lhd b \rhd q$ with p and q process terms and b a boolean condition behaves a p if b evaluates to true and as q otherwise.

Throughout the rest of this chapter, we adopt the convention that the $\cdot$ operator binds stronger than the $+$ operator. The conditional operator binds stronger than $+$ and weaker than $\cdot$.

When a system is made up of multiple components, these components will in general work alongside each other and communicate from time to time to influence the behaviour of the other components. To this end, process algebras also allow process terms to be put in parallel. The first parallel operator is the *merge* operator, denoted with $\|$, that represents two process terms working alongside each other. If two process terms $p = a$ and $q = b$, consisting of the execution of action a or b respectively, are put in parallel through a merge and no communication is possible between the two terms, the resulting process term will behave as the arbitrary interleaving of their actions. Hence, $p \parallel q$ behaves as $a \cdot b + b \cdot a$.

It may be the case, however, that the behaviour of certain actions in processes p and q is synchronised. To this end, the *communication* operator, denoted with $|$ is introduced. If we have two process terms $p = a$ and $q = b$, consisting of the execution of action a or b respectively, that are put in parallel through a merge operator as before, we may additionally specify that the actions a and b of these process terms synchronise: $a|b = c$. In this case, the parallel composition of the process terms will behave as the arbitrary interleaving of p and q, or expose their synchronised behaviour. Hence, $p \parallel q$ will behave as $a \cdot b + b \cdot a + c$.

To enforce that the actions of two processes may only occur synchronously, actions may be *encapsulated*. By encapsulating the actions a and b of the previous example, the parallel composition of p and q will only expose the synchronous behaviour; $p \parallel q$ will behave as c.

Encapsulation requires an additional action δ called deadlock. δ does not display any behaviour and is specified such that $p + \delta = p$ and $\delta \cdot p = \delta$. Encapsulation now works by substituting the atomic actions that make up a communication action with δ such that only the synchronous behaviour is exposed.

Process declarations

A process declaration is always of the form $P(x_1 : D_1, \ldots, x_n : D_n) = p$ with $n \geq 0$. It declares the process P that takes data variables $x_1 \ldots x_n$ as parameters and behaves as defined by the process term p. p may contain occurrences $Q(y_1, \ldots, y_n)$ that further specify the process to be executed. It may also contain a recursive call to P, as long as the recursive call is *guarded*, meaning that it is preceded by an action.

By using the *sum operator* $\sum_{d_1:D_1,\ldots,d_n:D_n} P(d_1, \ldots, d_n)$, a process term $P(d_1, \ldots, d_n)$ can be specified for any permutation of datum parameters $d_1 : D_1, \ldots, d_n : D_n$.

Finally, actions may be hidden through the use of the *hiding operator* τ_A. If this operator is applied to a process term p, all actions $a \in A$ will be substituted with the special action τ. This special action name is used for actions that are not observable or not of interest for the specification. Sometimes, however, the presence of τ actions can be observed as a result of the composition of process terms. In the process term $a + \tau \cdot b$, for example, the τ action is of interest; once it is executed, the set of possible behaviours of the system is reduced from $\{a + b, b\}$ to $\{b\}$. Such a τ-action is called *non-inert*. Conversely, an *inert* τ-action is an action that does not lose any possible behaviours.

For the sake of completeness, an overview of all axioms related to the notions discussed in this chapter are given in appendix B.

3.2 Model checking

One of the reasons to formally specify a program is to verify whether the program's behaviour is correct according to a specification. In this thesis, we will apply two techniques: proving the equivalence of a process to another process and model checking. In this section, we will discuss both notions, starting from a program specification in a process algebra as described in the previous section.

As a running example, we will use the following specification:

$$\begin{aligned} X &= a \cdot Y \\ Y &= b \cdot Y + c \cdot X \end{aligned}$$

of a program that executes infinitely many traces ab^*c. Figure 3.1 - (a) displays the automaton for this process.

(a) X (b) R

Figure 3.1: Automata of our running example X and its required external behaviour R

3.2.1 Process equivalence

The first approach to program verification is to show that the behaviour of a process conforms to its requirements. If these requirements can also be given in terms of processes, correctness of the program can be verified by checking whether the process specifying the behaviour is equivalent to the process specifying the requirements.

Over the years, many notions of process equivalence have been proposed. In [47, 46], many of these notions have been categorised into a hierarchy of identification. Two notions are of special interest as they form the extremes at either end of the hierarchy.

The first of these notions is trace equivalence. Two processes p and q are said to be trace equivalent if all of their possible executions are summarised by the same set of traces. This equivalence is located at the coarsest end of the spectrum, meaning that, given a process p and a set of other processes Q, it will find the most $q_n \in Q$ such that $q_n = p$.

At the other end of the spectrum, we find bisimulation equivalence, originally defined in [29]. In [47], it is defined as follows, based on the definition in [31]:

Definition 3.1. *A* bisimulation *is a binary relation R on processes, such that, for $a \in Act$*

1. *if pRq and $p \xrightarrow{a} p'$, then $\exists\, q' : q \xrightarrow{a} q'$ and $p'Rq'$;*
2. *if pRq and $q \xrightarrow{a} q'$, then $\exists\, p' : p \xrightarrow{a} p'$ and $p'Rq'$.*

with p, p', q, q' processes and Act the set of possible actions. Two processes p and q are said to be bisimilar, $p \leftrightarroweq q$, if there exists a bisimulation relation R such that pRq.

Bisimulation equivalence is located at the finest end of the spectrum, meaning that, given a process p and a set of other processes Q, it will find the least[6] $q_n \in Q$ such that $q_n = p$.

As an example of why these notions are different, consider the process terms:

$$\begin{aligned} p &= a \cdot b + a \cdot c \\ q &= a \cdot (b + c) \end{aligned}$$

for which the automata are shown in figure 3.2. The set of traces is the same for p and q: $\{a, b, c, ab, ac\}$. Hence, p and q are trace equivalent: $p =_T q$. However, p and q are not bisimilar: consider process q' obtained by taking the a-action in process q. Now, if p and q were to be bisimilar, there must be a process p' such that $p \xrightarrow{a} p'$ and $q' \leftrightarroweq p'$. Such a process does not exist. Hence $p \not\leftrightarroweq q$.

[6] Of course, the actual number of processes found to be equivalent may be the same for both notions depending on the characteristics of Q.

A refinement of bisimulation is *branching bisimulation* [48], which takes τ-transitions into account in the equivalence relation. Intuitively, inert τ transitions do not have to be performed by process p as well as q for p and q to be branching bisimilar $p \leftrightarrows_B q$. Branching bisimulation is defined as follows:

Definition 3.2. *A* branching bisimulation relation *is a binary relation R on processes, such that, for $a \in Act$*

1. *if pRq and $p \xrightarrow{a} p'$, then*
   (a) *either $a = \tau$ and $p'Rq$*
   (b) *or $\exists\, q'' : q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q''$ for zero or more τ transitions, such that $q'' \xrightarrow{a} q'$ with $p'Rq'$*
2. *if pRq and $q \xrightarrow{a} q'$, then*
   (a) *either $a = \tau$ and pRq'*
   (b) *or $\exists\, p'' : p \xrightarrow{\tau} \ldots \xrightarrow{\tau} p''$ for zero or more τ transitions, such that $p'' \xrightarrow{a} p'$ with $p'Rq'$*

with p, p', p'', q, q', q'' processes and Act the set of possible actions. Two processes p and q are said to be branching bisimilar, $p \leftrightarrows_B q$, if there exists a branching bisimulation relation R such that pRq.

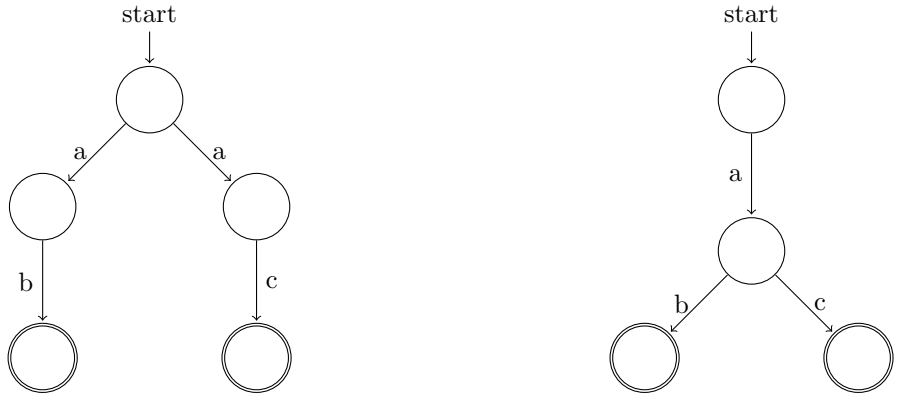
Branching bisimulation equivalence implicitly enforces a notion of *fairness* on processes when comparing them, to see if they are equivalent. Intuitively, this notion ensures that if an exit transition exists from a τ-loop, this transition will eventually be taken. Several fairness notions exist, for an overview we refer to [26].

Example of verification using process equivalence

As an example of the verification process using process equivalence, consider the example defined in the previous section, where we gave a specification for a program X that executes infinitely many traces ab^*c. Now, assume that the requirement for this program was to come up with a program that performs the action a followed by the action c infinitely many times. This requirement may be formalised as the process term:

$$R \quad = \quad a \cdot c \cdot R$$

for which the automaton is shown in figure 3.1 - (b).



(a) Process term p (b) Process term q

Figure 3.2: Process terms p and q, which are trace equivalent but not bisimilar

To prove that X conforms to the behavioural specification, we abstract away from all actions in X that are not of interest for the external behaviour by hiding them through use of the τ operator. As a result

of this, we now have a process specification X' in which all occurrences of the action b are replaced by the τ action. Then, we show that $X' \leftrightarrows_B R$.

To show that $X' \leftrightarrows_B R$, we first consider the initial states, labelled s_0 and s'_0 in figure 3.1. We see that both s_0 and s'_0 can perform an a-action and therefore, $s_0 \leftrightarrows_B s'_0$ by conditions 1b and 2b of definition 3.2. Now consider states s_1 and s'_1. First of all, both s_1 and s'_1 can perform a c-action to states s_0 and s'_0 respectively. In addition, s'_1 may perform a τ-action to itself. Hence, by by conditions 1b, 2a and 2b of definition 3.2, it follows that $s_1 \leftrightarrows_B s'_1$. By the fact that we already showed $s_0 \leftrightarrows_B s'_0$, we conclude that $X' \leftrightarrows_B R$.

Minimisation

As the complexity of processes increases, the size of the state space tends to grow exponentially. To battle this problem, several minimisation techniques have been proposed. One of these techniques is minimisation modulo branching bisimilarity, that prunes inert τ-transitions from a state space.

An efficient algorithm for minimisation modulo branching bisimilarity is proposed in [17]. The algorithm works by partitioning the state space in partitions $P_1 \cup \ldots \cup P_n$ through repeated application of the *split* operation:

Definition 3.3. *A state* $s_0 \in split_a(P_i, P_j)$ *for* $a \in Act \cup \{\tau\}$ *if* $\exists\, s_1 \ldots s_n$ *such that* $s_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$ *with* $s_0 \ldots s_{n-1} \in P_i$ *and* $s_n \in P_j$ *and* $n > 1$.

When there is no partition left on which the *split* operation can be executed to further split it, it holds for all branching bisimilar states s_0, s_1 that $s_0 \in P_i \wedge s_1 \in P_i$ for some partition P_i. Hence, if we have a partition with states $s \xrightarrow{\tau} s'$, the τ-transition can be pruned from the state space by collapsing s and s', without losing any behaviour. Note that by the fact that τ-transitions are pruned from the state space, a fairness assumption is again enforced on the state space. As a result of this, if two processes P and Q for which it holds that $P \leftrightarrows_B Q$ are both minimised modulo branching bisimilarity yielding processes P' and Q', it will hold that $P' \leftrightarrows Q'$.

3.2.2 Property checking

The other approach that we will apply to verify the correctness of a process is to formulate properties, and subsequently check that these properties hold on the state space that is generated for the process. In this technique, a distinction is made between *liveness* and *safety* properties.

A *liveness* property states that something 'good' will *eventually* happen. As an example of why liveness properties are useful, consider a traffic light of which initially the red light is burning. Obviously, the red light should eventually go out after which the green light should start burning for the traffic light to be useful. Therefore, a liveness property to check for this traffic light could be formulated as: *"if the red light is burning, eventually the green light must start burning"*.

A *safety* property states that something 'bad' will *never* happen. For the traffic light, it should not be the case that the red and green light are burning at the same time, as this may cause confusion which in turn may lead to accidents. Therefore, a safety property to check for the traffic light could be formulated as: *"the red and green light should never be burning at the same time"*.

Properties may be formulated using several different logics that differ in their expressive power such as LTL ([34]), CTL$^{(*)}$ ([11, 10]) or the μ-calculus ([25]). For this thesis, we will discuss the most expressive of these, the μ-calculus. Formulas in this calculus are defined by the following BNF grammar:

$$\phi \quad ::= \quad \mathtt{T} \mid \mathtt{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a]\phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

$\langle a \rangle \phi$ holds for a state s if there exists a state s' in which ϕ holds and $s \xrightarrow{a} s'$. $[a]\phi$ holds for a state s if for all transitions $s \xrightarrow{a} s'$, ϕ holds in s'. $\mathtt{T}$ holds in all states while $\mathtt{F}$ does not hold in any state. The set X ranges over recursion variables.

By the fact that μ-calculus formulas ϕ are monotonic, there exist minimal and maximal fixpoints $\mu X.\phi$ and $\nu X.\phi$. Here, ϕ represents a mapping that yields a set of states for which the property ϕ holds, ranging over the domain S of states in which the recursion variable X holds. To compute a minimal fixpoint $\mu X.\phi$, initially $S_0 = \emptyset$ is taken as a value for the recursion variable X. Repeatedly, the set S_{i+1} is computed consisting of those states in S_i for which ϕ holds. Eventually, $S_i = S_{i+1}$ – by monotonicity and the fact that there are only finitely many states – and a solution is found. To compute a maximal fixpoint $\nu X.\phi$, the set of all states is taken as a value for the recursion variable X.

As an extension to the μ-calculus, the regular μ-calculus was proposed ([28]) in which instead of formulas $\langle a\rangle\phi$ and $[a]\phi$, one may use formulas $\langle\beta\rangle\phi$ and $[\beta]\phi$ with β a regular expression defined by the following BNF grammar:

$$\begin{array}{lcl} \alpha & ::= & \mathtt{T} \mid a \mid \neg a \mid a \wedge a' \\ \beta & ::= & a \mid \beta \cdot \beta' \mid \beta|\beta' \mid \beta^* \end{array}$$

Here, α represents a set of actions, more specifically the set $\mathtt{T}$ of all actions, a the set containing a specific action $a \in \mathrm{Act} \cup \{\tau\}$, $\neg a$ its complement and $a \wedge a'$ the set of actions that occur both in a and a'.

β represents a set of traces consisting of a set of actions α, the concatenation $\cdot$ of a trace from β and β', the union $|$ of the traces in β and β' and finally β^* the transitive reflexive closure of β, consisting of those traces that a concatenation of finitely many traces from β yields.

$\langle\beta\rangle\phi$ is defined to hold if there is a trace β leading to a state s in which ϕ holds. $[\beta]\phi$ is defined to hold if all traces β end up in a state s where ϕ holds.

Example of verification using model checking techniques

As an example of the verification process using model checking techniques, recall the example in the previous section, where we specified a process X that executes infinitely many traces ab^*c. To prove that this process satisfies our requirement R, stating that it should perform the action a followed by the action c infinitely many times, we could verify that the following properties hold:

$$[(\neg a)^* \cdot c]\mathtt{F}$$

$$[\mathtt{T}^* \cdot a]\mu Y.(\langle\mathtt{T}\rangle\mathtt{T} \wedge [\neg c]Y)$$

$$[\mathtt{T}^* \cdot c]\mu Y.(\langle\mathtt{T}\rangle\mathtt{T} \wedge [\neg a]Y)$$

$$[\neg a \cdot \mathtt{T}^*]\mathtt{F}$$

The first property states that each c-action is preceded by an a-action. The second and third property state that each a-action is eventually followed by a c-action and vice versa. The final property states that all paths start with an a-action. Hence, by the last property, our program will first perform the a-action. By the second property it is eventually followed by a c-action. By the first and third property, this c-action will eventually be followed by an a-action before another c-action may occur *ad infinitum*.

The first, third and fourth property are easily proven to hold for our process X. The second property, however, poses a problem as the b action in state s_1 may be performed infinitely often. In fact, for this property to hold, we need to assume a fairness notion that ensures that while in state s_1 the c-transition, which is infinitely often enabled, is eventually taken.[7] To prove the property under this assumption, it can be reformulated as follows:

[7] Note that this fairness notion is different from the notion that we discussed previously, which was defined for τ-transitions.

$$[\mathtt{T}^* \cdot a \cdot (\neg c)^*]\langle(\neg c)^* \cdot c\rangle\mathtt{T}$$

In this formula, the latter part ($\langle(\neg c)^* \cdot c\rangle\mathtt{T}$) states that there exists a path consisting of arbitrarily many occurrences of an action other than c, followed by a c-action. By the first part of the formula ($[\mathtt{T}^* \cdot a \cdot (\neg c)^*]$), all of these paths are enabled after a path consisting of arbitrarily many actions ($\mathtt{T}^*$), followed by an a-action which is in turn followed by arbitrary many actions other than c ($(\neg c)^*$). This suffices to show that each a-action is eventually followed by a c-action under the assumption that the c-action of state s_1, which is infinitely often enabled, is eventually taken.

Summary

In this chapter, we introduced μCRL, a process algebra that distinguishes itself from others by its ability to handle data. We showed how to specify processes using a process algebra and discussed two model checking techniques: process equivalence and property checking. For both techniques, we gave an example application on a simple model, to show that this model satisfied requirements formulated for its behaviour.

In the next chapter, we will discuss our μCRL-specification of TCP in detail.

Chapter 4

Formal specification of TCP

This chapter discusses our formal specification of TCP using the process algebra μCRL. First, we will define the scope of our specification, after which we will discuss the data types that are important to our specification. We will then give a high-level overview of the model that we aim to specify, and discuss each of the components of this model in detail. Finally, we show how we combine these components to form a specification of a model of two TCP entities communicating over a possibly faulty medium.

4.1 Scope

When we set out to specify a protocol as complex as TCP, we were aware of the fact that the size of the state space could become unreasonably large. Therefore, we decided to focus our efforts on features that play a role during the data transfer phase and connection teardown. Hence, for the specification that is presented in this thesis, we assume a scenario where a connection with state `ESTABLISHED` has been set up but no data transfer has taken place yet; we pick up the execution of the protocol right after the completion of the three-way handshake.

We think that there are several arguments why it is reasonable not to consider the connection establishment procedure. First of all, many efforts have already been undertaken to formally specify and verify TCP's connection establishment. Second, the goal of our specification is to verify the correctness of the Window Scale Option. While this extension to the protocol does involve the agreement upon some parameters during connection establishment, this process is straightforward and its correctness can be derived manually. Hence, we do not see a reason to repeat earlier efforts to specify TCP's connection management only to include additional functionality that is highly unlikely to contain a mistake. Finally, we expect that if errors occur as a result of adding the Window Scale Option, they will become apparent during the data transfer phase or in the connection teardown procedure.

Even without including the connection establishment procedure, TCP is still a very large and complex protocol. Therefore, we decided to settle for a basic model, that only includes the features that are essential to achieve TCP's goal: reliable transfer over a possibly faulty medium. As a result of this decision, several aspects were excluded from our specification. This especially holds for some features as proposed in RFC 1122 [8]. For all features, we first of all considered whether we expected them to have an impact on the correctness of TCP extended with the Window Scale Option. The second criterion that we used was whether a proposed feature was essential for the correctness of the protocol – and therefore essential to include – or merely a performance enhancement.

Of the features in RFC 793 [36] that are related to data transfer or connection teardown, our specification as presented in this thesis does not include support for urgent data or the push function. The reason not to include support for the push function was that the de facto standard programming interface to TCP, the sockets API, does not include support for this function [43]. Likewise, in Berkeley-derived implementations the function is not needed because data delivery to the application layer is never delayed

[43]. The urgent data function is subject to much confusion over what its goal is. It is either used to relay out-of-band data (again in the sockets API), or to stimulate the application layer at the receiving end to issue the `RECEIVE` call. Either way, it is not of interest for the scope of our project as it does not alter the behaviour of (regular) data processing at the receiving end.

When a segment arrives at the receiver out of order (`SEG.SEQ` > `RCV.NXT`), it is not held for later processing but simply rejected and dropped. This means that our specification of TCP employs a go-back-n retransmission scheme. While this is not the optimal scheme in terms of performance, adding a feature that holds these segments for later processing would not greatly alter the behaviour of the protocol as these segments can also be seen as segments that have just arrived. Piggy-backing acknowledgements onto outgoing data segments, again a performance improvement, is also not included.

Any exceptional message processing that has to do with the reset flag is not specified since this flag only involves connection management issues: according to the specification, it should never be set if the connection is in the `ESTABLISHED` state.[8]

Finally, our model abstracts from corrupted segments since they do not add any behaviour to the protocol; if a data segment arrives at a TCP instance, a checksum is calculated and compared to the checksum as included in the header. If the checksums do not match, the segment is dropped. Hence, the behaviour of the protocol is no different than the behaviour for a lost segment. Adding a bit to the segments to identify whether a segment is corrupted or not would add unnecessary complexity to our state space. Therefore, analogous to [45], we abstract from corrupted segments in our specification.

Of the performance enhancing features in RFC 1122 [8] that are related to data transfer or connection teardown, our specification does not include the algorithms to avoid the Silly Window Syndrome as discussed on pages 89 and 97-100, improvements to the calculation of the retransmission timer (page 90), support for repackaging the segments on the retransmission queue (page 91) and the half-duplex close sequence, nor the reopening of a connection during the close sequence (page 88). We did include the corrections to the TCP connection state diagram related to the connection teardown procedure (page 86), the probing of zero windows (page 92), the acceptance criteria for incoming acknowledgements (page 94) and the remarks on when to send an acknowledgement segment (page 96).

4.2 Data types

Several data types had to be defined for our specification.

4.2.1 Booleans

First of all, `T` (true) and `F` (false) are introduced as boolean variables of data type *Bool*. In addition, we define several operations on booleans.

$$
\begin{array}{lcl}
\texttt{T} & : & \rightarrow \mathit{Bool} \\
\texttt{F} & : & \rightarrow \mathit{Bool} \\
\wedge & : & \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\
\vee & : & \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool}
\end{array}
\qquad
\begin{array}{lcl}
\neg & : & \mathit{Bool} \rightarrow \mathit{Bool} \\
= & : & \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\
\mathit{if} & : & \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool}
\end{array}
$$

Of these operations, $\wedge, \vee$ and $\neg$ are implemented as expected. $=$ defines equality of booleans while *if* defines an if-then-else construct on booleans:

[8] The RFC states one exception to this rule: *"if an incoming segment has a security level, or compartment, or precedence which does not exactly match the security level, or compartment, or precedence of the requested connection, a reset is sent and the connection goes to the CLOSED state."* However, security and precedence, while a feature, are not further specified and therefore not part of most real-world TCP implementations, except for implementations in a multilevel secure environment.

$$(x = y) = \begin{cases} \mathtt{T} & \text{if } x = \mathtt{T} \wedge y = \mathtt{T} \\ \mathtt{T} & \text{if } x = \mathtt{F} \wedge y = \mathtt{F} \\ \mathtt{F} & \text{otherwise} \end{cases} \qquad \mathit{if}(x, y, z) = \begin{cases} y & \text{if } x = \mathtt{T} \\ z & \text{if } x = \mathtt{F} \end{cases}$$

An operation $\mathit{if} \colon D \times D \to \mathit{Bool}$ is added in a similar fashion for any other data type D that we define.

4.2.2 Natural numbers

0 and S are introduced as constructors of data type Nat – the natural numbers – and several operations on natural numbers are defined:

0	:	$\to \mathit{Nat}$	$>$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$
S	:	$\mathit{Nat} \to \mathit{Nat}$	$\geq$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$
$=$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$	mod	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$
$+$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$	$\dot{-}$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$
$*$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$	*seq_diff*	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$
$\dot{:}$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$	*seq_between*	:	$\mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$
$<$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$	*seq_between_excl*	:	$\mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$
$\leq$	:	$\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$			

$=, +, *, <, \leq, >, \geq$ and mod are defined as one would expect them to be. $\dot{-}$ defines the *monus* operation ($-$ with the exception that it never goes below 0) while $\dot{:}$ is defined as integer division:

$$\begin{array}{lcl} 0 \dot{-} y & = & 0 \\ x \dot{-} 0 & = & x \\ S(x) \dot{-} S(y) & = & x \dot{-} y \end{array} \qquad x \mathbin{\dot{:}} y = \begin{cases} 1 + ((x - y) \mathbin{\dot{:}} y) & \text{if } y > 0 \wedge x \geq y \\ 0 & \text{if } y > 0 \wedge x < y \end{cases}$$

In addition to these standard operations, we had to define some operations on sequence numbers since sequence numbers are used modulo n, with n the size of the sequence number space. *seq_diff* defines a difference operation on sequence numbers, while *seq_between_excl* defines whether a sequence number x is between sequence numbers y and z. *seq_between* defines whether a sequence number x is between sequence numbers y and z or $x = z$.

$$\begin{array}{lcl} \mathit{seq\_diff}(0, y) & = & y \\ \mathit{seq\_diff}(S(x), 0) & = & \mathit{seq\_diff}(x, n \dot{-} 1) \\ \mathit{seq\_diff}(S(x), S(y)) & = & \mathit{seq\_diff}(x, y) \\ \mathit{seq\_between}(x, y, z) & = & \mathit{seq\_between\_excl}(x, y, z) \vee x = z \\ \mathit{seq\_between\_excl}(x, 0, y) & = & x < y \\ \mathit{seq\_between\_excl}(S(x), S(y), 0) & = & \mathit{seq\_between\_excl}(x, y, n \dot{-} 1) \\ \mathit{seq\_between\_excl}(0, S(y), S(z)) & = & \mathit{seq\_between\_excl}(n - 1, y, z) \\ \mathit{seq\_between\_excl}(0, S(y), 0) & = & \mathit{seq\_between\_excl}(n - 1, y, n - 1) \\ \mathit{seq\_between\_excl}(S(x), S(y), S(z)) & = & \mathit{seq\_between\_excl}(x, y, z) \end{array}$$

4.2.3 Segments

Of the TCP header, only the sequence number, the acknowledgement number, the window size, the `ACK` flag and the `FIN` flag are important to our specification. In addition, we need to know the number of octets that are included in the segment as data. This can normally be calculated by subtracting the data offset from the length field that is included in the IP header. In our specification, however, we will abstract away from this implementation detail and simply include this number in the segment. Note that

our segments do not contain a buffer with data, as such a buffer can easily be reconstructed from the information in the header. Hence, a data type *Sgmt* representing segments is defined with the following constructor function:

$$sgmt \quad : \quad Nat \times Nat \times Nat \times Nat \times Bool \times Bool \rightarrow Sgmt$$

In addition, we define the following operations on segments:

$=$	:	$Sgmt \times Sgmt \rightarrow Bool$	*get_num_octs*	:	$Sgmt \rightarrow Nat$
get_seq_nr	:	$Sgmt \rightarrow Nat$	*is_acknowledgement*	:	$Sgmt \rightarrow Bool$
get_ack_nr	:	$Sgmt \rightarrow Nat$	*fin_flag_set*	:	$Sgmt \rightarrow Bool$
get_window	:	$Sgmt \rightarrow Nat$			

Two segments are defined to be equal, expressed by the operation $=$, if all of their arguments are equal. *get_seq_nr*, *get_ack_nr*, *get_window*, *get_num_octs*, *is_acknowledgement* and *fin_flag_set* are destructor methods that return the first, second, third, fourth, fifth and sixth argument of a segment, respectively.

4.2.4 Octet buffers

A TCP instance maintains a send and receive buffer for octets that have either just been accepted from the application layer and are waiting to be sent or that have just arrived and are ready to be forwarded to the application layer. To reduce the size of the state space, these buffers are implemented as sorted lists of natural numbers. For such lists, we defined the data type *Buffer* with constructor functions $\emptyset$ and *li*, for the empty list and an item in the list respectively. Additionally, we specify several operations on buffers:

$\emptyset$	:	$\rightarrow Buffer$			
li	:	$Nat \times Buffer \rightarrow Buffer$	$\in$	:	$Nat \times Buffer \rightarrow Bool$
$=$	:	$Buffer \times Buffer \rightarrow Bool$	*merge*	:	$Buffer \times Buffer \rightarrow Buffer$
first	:	$Buffer \rightarrow Nat$	*take_set*	:	$Buffer \times Buffer \rightarrow Buffer$
rest	:	$Buffer \rightarrow Buffer$	*infl*	:	$Nat \times Nat \rightarrow Buffer$
add	:	$Nat \times Buffer \rightarrow Buffer$	*add_ordered*	:	$Nat \times Buffer \rightarrow Buffer$
length	:	$Buffer \rightarrow Nat$	*take_n*	:	$Buffer \times Nat \rightarrow Buffer$

$=$ defines equality of two buffers as the equality of their contents. *first* returns the first element in a buffer, while *rest* returns the buffer without its first element. *add* prepends an element to the front of the buffer. Likewise *add_ordered* adds an element to the buffer, but ensures that the buffer stays ordered. *length* returns the number of elements in the buffer. *merge* takes two buffers as arguments and returns an ordered buffer that is the result of merging the two. If an element x occurred in both buffers, it will occur twice in the resulting buffer. *take_n* takes one occurrence of an element x from a buffer. *take_set* takes two buffers b_1 and b_2 as arguments, and returns a buffer b_3 that consists of the buffer b_1 to which *take_n* is applied for every element in b_2. Finally, *infl* takes two sequence numbers x and y as arguments, and yields an ordered buffer that contains y sequence numbers starting at x (taking the fact that sequence numbers are taken modulo n into account).

$$
\begin{array}{lcl}
\emptyset = \emptyset & = & \mathtt{T} \\
li(x, b_1) = li(y, b_2) & = & x = y \wedge b_1 = b_2 \\
\mathit{first}(li(x, b_1)) & = & x \\
\mathit{rest}(\emptyset) & = & \emptyset \\
\mathit{rest}(li(x, b_1)) & = & b_1 \\
\mathit{add}(x, b_1) & = & li(x, b_1) \\
\mathit{length}(\emptyset) & = & 0 \\
\mathit{length}(li(x, b_1)) & = & 1 + \mathit{length}(b_1) \\
x \in \emptyset & = & \mathtt{F} \\
x \in li(y, b_1) & = & x = y \vee x \in b_1
\end{array}
\qquad
\begin{array}{lcl}
\mathit{merge}(\emptyset, b_1) & = & b_1 \\
\mathit{merge}(b_1, \emptyset) & = & b_1 \\
\mathit{merge}(li(x, b_1), b_2) & = & \mathit{add\_ordered}(x, \mathit{merge}(b_1, b_2)) \\
\mathit{take\_set}(b_1, \emptyset) & = & b_1 \\
\mathit{take\_set}(\emptyset, b_1) & = & \emptyset \\
\mathit{take\_set}(b_1, li(x, b_2)) & = & \mathit{take\_set}(\mathit{take\_n}(b_1, x), b_2) \\
\mathit{infl}(x, 0) & = & \emptyset \\
\mathit{infl}(x, S(y)) & = & \mathit{add\_ordered}(x, \mathit{infl}((x+1) \bmod n, y) \\
\mathit{take\_n}(\emptyset, x) & = & \emptyset
\end{array}
$$

$$
\mathit{take\_n}(li(x, b_1), y) = \begin{cases} li(x, \mathit{take\_n}(b_1, y)) & \text{if } x < y \\ b_1 & \text{if } x = y \\ li(x, b_1) & \text{otherwise} \end{cases}
$$

$$
\mathit{add\_ordered}(x, li(y, b_1)) = \begin{cases} li(x, li(y, b_1)) & \text{if } x \leq y \\ li(y, \mathit{add\_ordered}(x, b_1)) & \text{otherwise} \end{cases}
$$

4.2.5 Segment buffers

In addition to a buffer for octets that have just been accepted from the application layer and are waiting to be sent, the sender also maintains a retransmission queue. Furthermore, mediums may hold any number of segments at any point in time. To facilitate these requirements, we have defined *SgmtQueue*, a buffer of segments, with constructor functions $\emptyset$ and *qu*, for the empty queue and a segment on the queue respectively. On these queues, we again define several operations:

$$
\begin{array}{lcl}
\mathit{empq} & : & \rightarrow \mathit{SgmtQueue} \\
\mathit{qu} & : & \mathit{Sgmt} \times \mathit{SgmtQueue} \rightarrow \mathit{SgmtQueue} \\
\mathit{first} & : & \mathit{SgmtQueue} \rightarrow \mathit{Sgmt} \\
\mathit{rest} & : & \mathit{SgmtQueue} \rightarrow \mathit{SgmtQueue} \\
\mathit{add} & : & \mathit{Sgmt} \times \mathit{SgmtQueue} \rightarrow \mathit{SgmtQueue}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{add\_ordered} & : & \mathit{Sgmt} \times \mathit{SgmtQueue} \rightarrow \mathit{SgmtQueue} \\
\mathit{remove} & : & \mathit{Sgmt} \times \mathit{SgmtQueue} \rightarrow \mathit{SgmtQueue} \\
\mathit{length} & : & \mathit{SgmtQueue} \rightarrow \mathit{Nat} \\
= & : & \mathit{SgmtQueue} \times \mathit{SgmtQueue} \rightarrow \mathit{Bool} \\
\in & : & \mathit{SgmtQueue} \rightarrow \mathit{Bool}
\end{array}
$$

The operations *first*, *rest*, *add*, *add_ordered*, *length*, $=$ and $\in$ are defined in a fashion similar to the operations on *Buffer* as discussed in the previous section. *remove* takes a segment s and a buffer b_1 as a parameter and returns b_1' without s.

4.2.6 Connection states

Since a connection may progress through several states, a data type *ConnectionState* is defined, with the following constructors:

$$
\begin{array}{lcl}
\mathit{CLOSED} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{LISTEN} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{SYN\_SENT} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{SYN\_RECEIVED} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{ESTABLISHED} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{FIN\_WAIT\_1} & : & \rightarrow \mathit{ConnectionState}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{FIN\_WAIT\_2} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{CLOSE\_WAIT} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{CLOSING} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{LAST\_ACK} & : & \rightarrow \mathit{ConnectionState} \\
\mathit{TIME\_WAIT} & : & \rightarrow \mathit{ConnectionState}
\end{array}
$$

Apart from the equality function = and conditional function *if*, no other functions have been defined for this data type. Additionally, we specified a data type *ConnectionStates* representing a list of connection states to achieve notational convenience in the guards of our specification, where we can compare the current state using the $\in$ operator over a list of states rather than a conjunction of states that are applicable for the guard.

4.2.7 The Transmission Control Block

The Transmission Control Block (TCB) maintains all variables for a connection. It is specified as data type *TransmissionControlBlock*, with the following constructor:

$$tcb \quad : \quad \begin{aligned} &Buffer \times SgmtQueue \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Buffer \times \\ &Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Bool \rightarrow TransmissionControlBlock \end{aligned}$$

The TCB contains the following variables (listed in the order of the constructor) :

Variables related to sending segments

- The send buffer
- The retransmission queue
- `SND_UNA`
- `SND_NXT`
- `SND_WND`
- `SND_UP`
- `SND_WL1`
- `SND_WL2`
- `SND_ISS`†
- `SND_WND_SCALE`†

Variables related to receiving segments

- The receive buffer
- `RCV_NXT`
- `RCV_WND`†
- `RCV_UP`
- `RCV_IRS`†
- `RCV_WND_SCALE`†
- `RCV_RD_NXT`
- `RCV_ACK_QUEUED`

We refer to chapter 2 for a detailed explanation of all variables except `RCV_RD_NXT` and `RCV_ACK_QUEUED`. `RCV_RD_NXT` keeps track of the next octet that the TCP instance should forward to the application layer, while `RCV_ACK_QUEUED` is used to determine whether an acknowledgement should be sent to the remote entity to acknowledge the reception of a segment. For each of the variables in the TCB, a getter and setter function are defined. Furthermore, equality of TCBs tcb_1 and tcb_2 is defined as equality of all variables of tcb_1 and tcb_2.

For the variables marked with a †, the value is determined during connection establishment and remains constant during the execution of the protocol. We chose to include them in the TCP for future extensibility; if we wish to add connection establishment to our specification at a later stage, the variables are already included. The same holds for the variables `SND_UP` and `RCV_UP`, which are not used currently since we have not included the urgent function in our specification. Also note that, in case of a unidirectional connection, variables related to sending segments will remain constant during the execution of the protocol at the sending TCP instance while variables related to receiving segments will remain constant at the receiving TCP instance.

4.2.8 Utility functions

Apart from the data types and functions as discussed above, we have also implemented several utility functions. First of all, *can_send* is used by a TCP instance to determine the number of octets that it

may send. *rcv_wnd* is used to calculate the size of the receive window that should be advertised to the remote entity. *adv_wnd* also calls this method and subsequently applies the scale factor to its result by means of the operator $\dot{\div}$. *may_accept_ack* determines whether an acknowledgement segment is acceptable while *must_update_window* determines whether the window information in an acknowledgement is 'fresh' enough to be used to update the size of the send window. *receiver_may_accept* is used to determine whether an incoming segment is eligible to be accepted. Finally, *update_rtq* is used to remove segments from the retransmission queue that are acknowledged by an incoming acknowledgement and *construct_ack* takes the current state of the TCB as a parameter and yields an acknowledgement segment.

The signature of these functions is given below. Their implementation is discussed in the remainder of this chapter at the appropriate places.

$$
\begin{array}{lcl}
\mathit{can\_send} & : & \mathit{TransmissionControlBlock} \to \mathit{Nat} \\
\mathit{rcv\_wnd} & : & \mathit{TransmissionControlBlock} \to \mathit{Nat} \\
\mathit{adv\_wnd} & : & \mathit{TransmissionControlBlock} \to \mathit{Nat} \\
\mathit{may\_accept\_ack} & : & \mathit{Sgmt} \times \mathit{TransmissionControlBlock} \to \mathit{Bool} \\
\mathit{must\_update\_window} & : & \mathit{Sgmt} \times \mathit{TransmissionControlBlock} \to \mathit{Bool} \\
\mathit{receiver\_may\_accept} & : & \mathit{Sgmt} \times \mathit{TransmissionControlBlock} \to \mathit{Bool} \\
\mathit{update\_rtq} & : & \mathit{SgmtQueue} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{SgmtQueue} \\
\mathit{construct\_ack} & : & \mathit{TransmissionControlBlock} \to \mathit{Sgmt}
\end{array}
$$

4.3 Specification

Figure 4.1 shows a conceptual overview of our specification. It consists of two TCP instances (*TCP1* and *TCP2*) that communicate over a possibly faulty medium. Since our specification of a medium is unidirectional, our system contains a medium for segments that are transferred from *TCP1* to *TCP2* (*NL1*) and a medium for segments that are transferred from *TCP2* to *TCP1* (*NL2*). For each of the TCP instances, an application layer is added to the system. The separate components communicate with each other over communication channels using a synchronous message passing scheme, represented by the synchronous calling of two actions that are defined to synchronise.

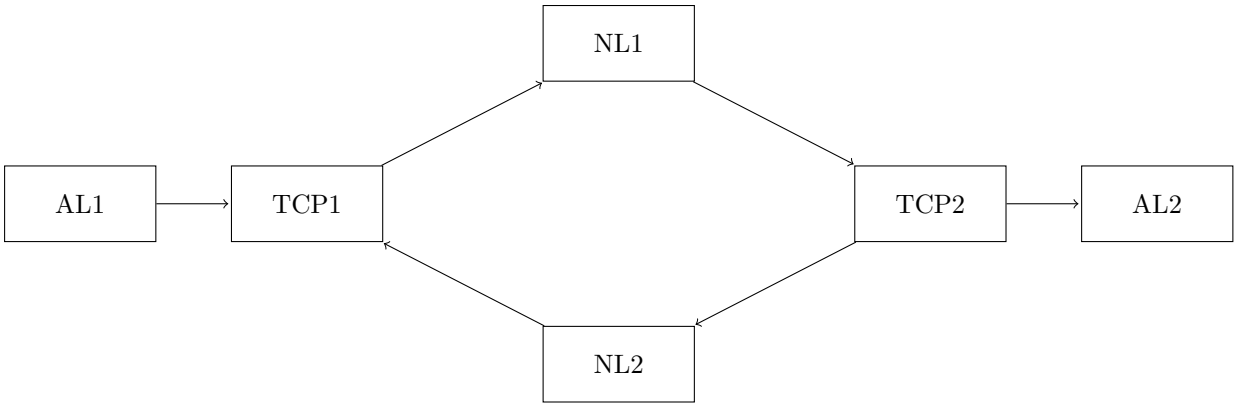


Figure 4.1: Conceptual overview of our model

We will now first discuss each of the components in detail, after which we will show how they can be combined to form our model.

4.3.1 The application layer

[36] distinguishes between five calls that may be issued from the application layer to TCP: `SEND`, `RECEIVE`, `CLOSE`, `ABORT` and `STATUS`. `SEND` calls are issued to TCP to supply it with data that the application layer

wishes to transfer to the remote entity. `RECEIVE` calls are issued in order to receive data coming from the remote entity. The `CLOSE` call causes the TCP connection to be closed. Our specification of the application layer includes these three calls.

`ABORT` is not included in our specification, because it is only issued in case of errors in the connection establishment procedure, which we have not modelled in the first place, or the connection termination procedure. In the latter case, the `ABORT` call *may* be issued if the TCP instance is in the `LAST_ACK` state and an acknowledgement is not received within a user-specified timeout period, a situation that may occur if the remote entity crashed. Since our specification does not include this timeout period, this behaviour gives us no reason to specify the `ABORT` call.

The `STATUS` call is also not included, because it does not influence the functionality of TCP. [36] even states: *"this is an implementation dependent user command and could be excluded without adverse effect"*. Figure 4.2 shows a conceptual overview of our application layer.

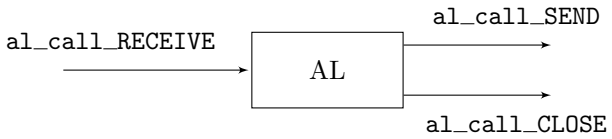


Figure 4.2: Conceptual overview of the application layer and its channels

To specify the `SEND` call appropriately, we assume that the application layer continuously offers octets to the TCP instance by issuing the call `al_call_SEND`. With each call, one octet is offered to TCP. The octets are represented as natural numbers taken from a fixed set of natural numbers. The number is wrapped back to 0 once the octet numbered $n-1$ has been read. The parameter x represents the sequence number of the next octet that the application layer will offer to TCP, while the parameter y represents the total number of octets that the application layer will offer to TCP.

This behaviour is slightly different from a real-world implementation in two aspects. First of all, the application layer may offer a buffer of octets to TCP according to the specification. Such a buffer is only accepted by TCP if enough space is available in the send buffer. We think that offering one octet at a time is only a slight deviation from this behaviour and that the difference can be seen as a low-level detail. Furthermore, the application layer would in reality emit octets that are only mapped to a sequence number when they are put in TCP's send buffer. We are, however, not interested in the actual data but only in the order in which the data is delivered and therefore abstract away from the actual octet values.

Receiving data is modelled by having the application layer call `al_call_RECEIVE` for an arbitrary octet. Since octets are represented as natural numbers taken from a fixed set of size n, this call may be issued for any natural number $z < n$.

Finally, [36] states that a TCP instance issues the `CLOSE` call if it has no more data to send. Therefore, once y reaches 0, the application layer call `al_call_CLOSE` may be issued.

Taken together, this leads to the following specification:

$$\begin{aligned} AL(x{:}Nat, y{:}Nat) \quad &= \quad al\_call\_SEND(x) \cdot AL((x+1) \bmod n, y-1) \ \lhd\ y > 0 \ \rhd\ \delta \\ &+ \quad \sum_{z:Nat} \Big(al\_call\_RECEIVE(z) \cdot AL(x,y) \ \lhd\ z < n \ \rhd\ \delta \Big) \\ &+ \quad al\_call\_CLOSE \cdot AL(x,y) \ \lhd\ y = 0 \ \rhd\ \delta \end{aligned} \tag{4.1}$$

Note that the application layer does not change state after it has accepted the `RECEIVE` call, as we abstract from the actual processing of the data at the application layer.

4.3.2 The TCP instance

The TCP protocol manages all communications of an entity with remote parties. If we have two entities A and B, several communications may be ongoing between them for applications $a_1 \dots a_n$ and $b_1 \dots b_m$. Each of these communications (a_x, b_y) with $x \leq n, y \leq m$ is referred to as a connection, which is identified by the unique network addresses of the hosts and the port numbers of the applications that engage in the communication. Within a connection, an entity may act both as a sender and receiver of octets.

The specification of a TCP instance that we describe here manages a single connection that we assume to be established. In a sense, what we have modelled here is a TCP connection rather than a TCP instance. However, to avoid discussing abstract notions such as connections (at the TCP level) and sessions (at the application level) that are rather detached from their contexts of sending and receiving entities in a network, we will take the point of view that we have modelled a TCP instance which only has one connection with one remote entity. This TCP instance maintains a variable s that holds the state that its connection is in, and a variable t that holds the transmission control block (TCB). For an overview of the parameters that are kept in the TCB and the transitions between states, we refer to our functional specification of TCP.

In the discussion of our specification of a TCP instance, we mostly follow the structure of [36]. To ease understanding of our specification, figures 4.4 and 4.5 show an abstract overview of our specification of TCP's data transfer phase while figure 2.3 shows an overview of the connection teardown procedure.

The responsibilities of TCP can be categorised into two categories: responding to calls from the application layer and responding to events that occur. We will discuss each of these categories in detail.

Calls from the application layer

For each of the channels `al_call_*` that we specified in the application layer, a channel `tcp_rcv_*` is specified at the TCP instance. Furthermore, we specify the channels `tcp_CALL_SND` and `tcp_call_RECV` that model the communications of TCP with the network layer. In the next chapter, we will show how these actions are forced into communication when the separate components of our model are composed. Figure 4.3 shows a conceptual overview of the TCP instance as we have modelled it.

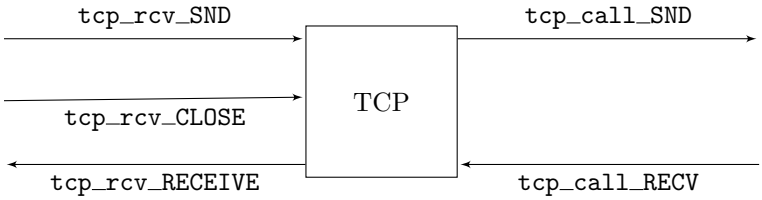


Figure 4.3: Conceptual overview of the TCP instance and its channels

The first call that is discussed in [36] is the `SEND` call on pages 56 to 57. By issuing a `tcp_rcv_SND` call the TCP instance accepts an arbitrary octet and adds it to its send buffer.[9] The call may only be issued if the connection is in state `ESTABLISHED` or `CLOSE_WAIT` and if there is send buffer space available for the octet:

[9]Note that in our specification, all updates to variables in the TCB are of the form $a \mapsto b$, meaning that variable a of the TCB is replaced by value b. If a is also used at the right side of the substitution, this means that the old value is first retrieved from the TCB for manipulation.

$$
\begin{aligned}
&TCP(s{:}ConnectionState, t{:}TransmissionControlBlock) = \\
&\quad \sum_{x:Nat} \Big(tcp\_rcv\_SND(x) \cdot TCP\Big(s, t\big[send\_buffer \mapsto add\_ordered(x, send\_buffer)\big]\Big) \\
&\quad \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT\} \\
&\qquad\qquad \wedge x < n \wedge length(get\_send\_buffer(t)) < buffer\_capacity\ \rhd\ \delta\Big)
\end{aligned}
\tag{4.2}
$$

In addition, [36] specifies that TCP may segmentise octets in the send buffer and subsequently send them to the remote entity *"at its own will"*. To this end, we include the following summand in our specification:

$$
\begin{aligned}
+\quad & tcp\_call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}))\cdot \\
& TCP\Bigg(s, t\Big[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, adv\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}), rtq), \\
& \qquad\qquad send\_buffer \mapsto take\_set(send\_buffer, infl(SND\_NXT, can\_send(t))), \\
& \qquad\qquad SND\_NXT \mapsto (SND\_NXT + can\_send(t)) \bmod n\Big]\Bigg) \\
& \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT\} \wedge can\_send(t) > 0\ \rhd\ \delta
\end{aligned}
\tag{4.3}
$$

If the connection is in the state `ESTABLISHED` or `CLOSE_WAIT` and TCP is allowed to send one or more octets, a segment containing the octets that are eligible to be sent is passed to the network layer by issuing the call `tcp_call_SND`. This segment is labelled with the next sequence number to be used as maintained in the TCB variable `SND_NXT`. After the sequence number, the acknowledgement number and advertised window are included, followed by the number of octets included in the segment and the values of the `ACK` and `FIN` flag. The size of the receive window is calculated through the method *rcv_wnd* as follows:

$$
rcv\_wnd(t) = \big(\mathtt{get\_RCV\_WND}(t) - seq\_diff(\mathtt{get\_RCV\_RD\_NXT}(t), \mathtt{get\_RCV\_NXT}(t))\big) \bmod n
$$

The function *adv_wnd* calls *rcv_wnd* and subsequently applies the window scale factor:

$$
adv\_wnd(t) = rcv\_wnd(t) \mathbin{\dot{-}} \mathtt{get\_RCV\_WND\_SCALE}(t)
$$

Subsequently, the octets included in the segment are removed from the send buffer and the segment is added to the retransmission queue (*rtq*). Finally, `SND_NXT` is updated to reflect the next sequence number to be used.

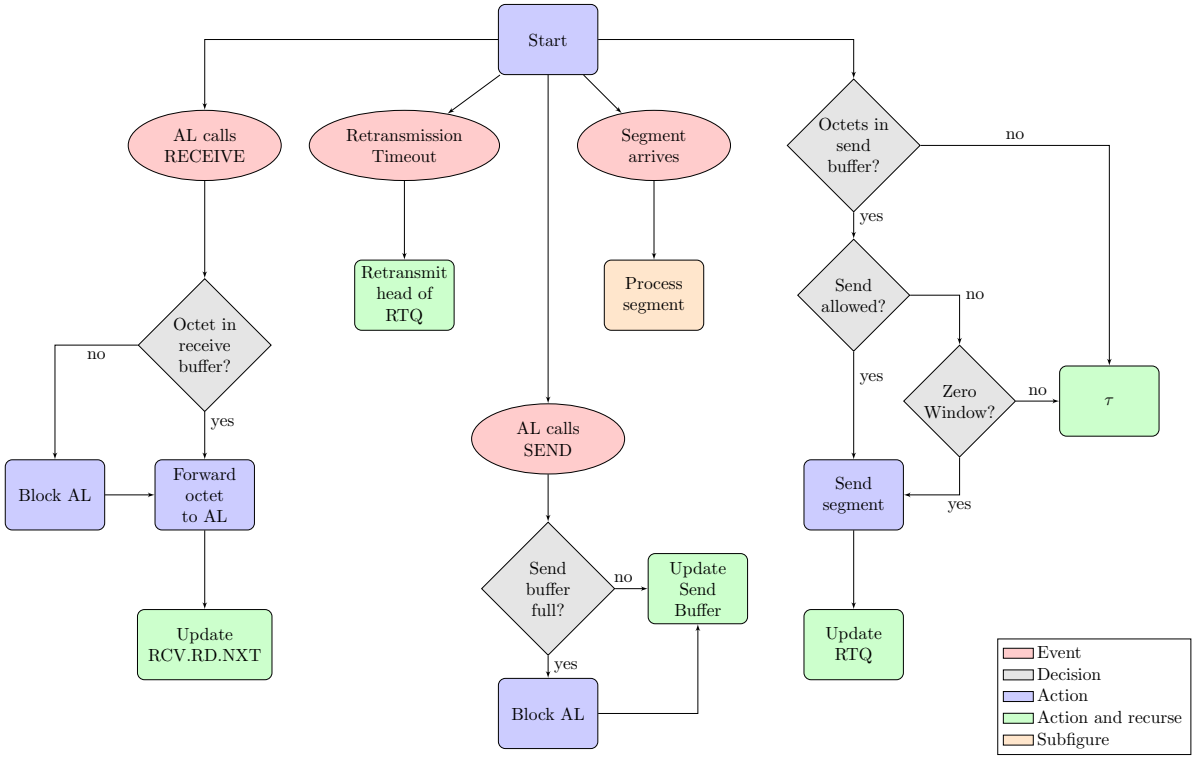


Figure 4.4: Abstract overview of our specification of TCP Data Transfer

In our model, the `ACK` flag will always be set to false in segments carrying data octets and therefore, the value of the acknowledgement field in the segment will not be processed by the receiver of the segment. The specification dictates that the `ACK` flag is always set to true and that the latest acknowledgement information is always included in each data segment. However, this would greatly complicate the processing of incoming segments in our model and would only be of use in case of a bidirectional connection, that we do not consider to limit the size of our state space. In a unidirectional setting, the sender's value of `RCV_NXT` will always be the same since it never receives data. Likewise, the receiver's values of `SND_NXT` and `SND_UNA` will remain constant since it never sends data. Hence, at all times during the execution of the protocol, if we have a sender A and a receiver B that have agreed on initial sequence number x, it will hold that $A_{\texttt{RCV\_NXT}} = B_{\texttt{SND\_NXT}} \wedge B_{\texttt{SND\_NXT}} = B_{\texttt{SND\_UNA}} = x$ and therefore, acknowledgement processing will not take place (since $\neg(B_{\texttt{SND\_UNA}} < A_{\texttt{RCV\_NXT}})$). For this reason, we chose to exclude this behaviour from our model. It can, however, easily be added by setting the `ACK` flag to true in all cases and adapting the processing of incoming segments to first consider acknowledgement information and then data and control information, if any. Even then, this would only lead to improved performance in terms of number of sent segments.

A little more can be said about the octets that are eligible to be sent. The function *can_send* first checks whether $\texttt{SND\_UNA} \leq \texttt{SND\_NXT} < (\texttt{SND\_UNA} + \texttt{SND\_WND})$. If this is the case, the sender is allowed to send $x = (\texttt{SND\_UNA} + \texttt{SND\_WND}) - \texttt{SND\_NXT}$ octets. The function will return x if the length of the buffer is greater than x or otherwise the length of the buffer. If $\neg(\texttt{SND\_UNA} \leq \texttt{SND\_NXT} < (\texttt{SND\_UNA} + \texttt{SND\_WND}))$, the function returns 0, indicating that no octets may be sent by TCP.

For the next summand, it is important to recall that in each acknowledgement, a receiving TCP instance may adjust the size of the send window of the remote entity. Now suppose that the receiving entity has adjusted the send window of its remote entity to a size of 0. This may lead to a deadlock, since the sender is not allowed to send anything and therefore will not receive any additional acknowledgements that may contain an updated window size. Therefore, a TCP instance must regularly transmit something to the remote entity if the variable `SND_WND` is set to 0. This is specified as follows:

$$
\begin{aligned}
+ \quad & \mathit{tcp\_call\_SND}(\mathit{sgmt}(\mathit{get\_SND\_NXT}(t), \mathit{get\_RCV\_NXT}(t), \mathit{adv\_wnd}(t), 1, \mathtt{F}, \mathtt{F}))\cdot \\
& \mathit{TCP}\Big(s, t\Big[\mathit{rtq} \mapsto \mathit{add}(\mathit{sgmt}(\mathit{SND\_NXT}, \mathit{RCV\_NXT}, \mathit{adv\_wnd}(t), 1, \mathtt{F}, \mathtt{F}), \mathit{rtq}), \\
& \qquad \mathit{send\_buffer} \mapsto \mathit{take\_set}(\mathit{send\_buffer}, \mathit{infl}(\mathit{SND\_NXT}, 1)), \\
& \qquad \mathit{SND\_NXT} \mapsto (\mathit{SND\_NXT} + 1) \bmod n\Big]\Big) \\
& \lhd\ \mathit{can\_send}(t) = 0 \wedge \mathit{get\_SND\_WND}(t) = 0 \\
& \quad \wedge \mathit{length}(\mathit{get\_rtq}(t)) = 0 \wedge \mathit{length}(\mathit{get\_send\_buffer}(t)) > 0 \ \rhd\ \delta
\end{aligned}
\tag{4.4}
$$

If the send window is 0 and the retransmission queue is empty, but octets are available in the send buffer, the sender will construct a segment containing one octet and send it. Again, the octet included in the segment is taken from the send buffer, the segment is put on the retransmission queue and the variable `SND_NXT` is updated.

The second call from the application layer that the TCP instance must process is the `RECEIVE` call, as discussed on pages 58-59 of [36]. By issuing a `tcp_rcv_RECEIVE` call, parameterised with the octet pointed at by the `RCV_RD_NXT` variable as maintained in the TCB, that octet is offered to the application layer in response to the `RECEIVE` call. Furthermore, the octet is removed from the receive buffer and the variable `RCV_RD_NXT` is incremented. The `tcp_rcv_RECEIVE` call may only be issued if the connection is in the state `ESTABLISHED`, `FIN_WAIT_1`, `FIN_WAIT_2` or `CLOSE_WAIT`, (`RCV_NXT` − `RCV_RD_NXT`) mod $n > 0$ and the octet with the sequence number stored in `RCV_RD_NXT` is available in the receive buffer.

$$
\begin{aligned}
+ \quad & \mathit{tcp\_rcv\_RECEIVE}(\mathit{get\_RCV\_RD\_NXT}(t))\cdot \\
& \mathit{TCP}\Big(s, t\Big[\mathit{receive\_buf} \mapsto \mathit{take\_n}(\mathit{receive\_buf},\ \mathit{RCV\_RD\_NXT}) \\
& \qquad \mathit{RCV\_RD\_NXT} \mapsto (\mathit{RCV\_RD\_NXT} + 1) \bmod n\Big]\Big) \\
& \lhd\ s \in \{\mathit{ESTABLISHED},\ \mathit{FIN\_WAIT\_1},\ \mathit{FIN\_WAIT\_2},\ \mathit{CLOSE\_WAIT}\} \\
& \quad \wedge \mathit{seq\_diff}(\mathit{get\_RCV\_RD\_NXT}(t), \mathit{get\_RCV\_NXT}(t)) > 0 \\
& \quad \wedge \mathit{length}(\mathit{get\_receive\_buf}(t)) > 0 \\
& \quad \wedge \mathit{get\_RCV\_RD\_NXT}(t) \in \mathit{get\_receive\_buf}(t) \ \rhd\ \delta
\end{aligned}
\tag{4.5}
$$

The variable `RCV_RD_NXT` is not mentioned in [36]. Instead, the size of the receive window, stored as `RCV_WND` in the TCB, is updated every time the receive buffer is manipulated. However, on page 74 a strict requirement is formulated that the total of `RCV_WND` and `RCV_NXT` must not be reduced. To simplify the implementation but ensure that this requirement is not violated, we chose to maintain the value for the variable `RCV_WND` at its initial value at all times, and introduce the variable `RCV_RD_NXT` that always contains the sequence number of the next octet that TCP will forward to the application layer. Hence, at all times it holds that `RCV_NXT` ≤ `RCV_RD_NXT` ≤ (`RCV_NXT` + `RCV_WND`). Rather than updating `RCV_WND`, the window size that should be reported back to the remote entity in acknowledgement segments - the advertised window - can be calculated from `RCV_WND`, `RCV_RD_NXT` and `RCV_NXT` instead, through the

rcv_wnd function as discussed before. Implicitly, this limits the maximum capacity of the receive buffer to the initial size taken for the receive window. We believe that in real-world implementations, this maximum capacity may be higher. However, [36] assumes the size of the receive window chosen during connection establishment to be equal to the capacity of the receive buffer. Moreover, as it is an edge case, this is the situation that is of interest for a correctness verification.

Finally, the TCP instance may receive a `CLOSE` call from the application layer, as discussed on pages 60-61 of [36]. The specification states that if such a call is issued by the application layer while there are still octets in the send buffer, TCP will queue this call until all of these octets are segmentised. Hence, the TCP instance may only perform the `tcp_rcv_CLOSE` action whenever its buffer is empty. The call is processed by sending a segment with the `FIN` flag set, after which the TCP instance will either progress to the `FIN_WAIT_1` or `LAST_ACK` state. The first case models the situation where the connection is still fully opened, while the second case conforms to the situation where the TCP instance has received a `FIN` segment, signalling that the other end has closed the connection.

$$
\begin{aligned}
&+ \quad \mathit{tcp\_rcv\_CLOSE} \cdot \\
&\quad \mathit{tcp\_call\_SND}(\mathit{sgmt}(\mathit{get\_SND\_NXT}(t), \mathit{get\_RCV\_NXT}(t), \mathit{adv\_wnd}(t), 0, \mathtt{F}, \mathtt{T})) \cdot \\
&\quad \mathit{TCP}\Big(\mathit{FIN\_WAIT\_1}, t\Big[\mathit{rtq} \mapsto \mathit{add}(\mathit{sgmt}(\mathit{SND\_NXT}, \mathit{RCV\_NXT}, \mathit{adv\_wnd}(t), 0, \mathtt{F}, \mathtt{T}), \mathit{rtq}), \\
&\qquad\qquad \mathit{SND\_NXT} \mapsto (\mathit{SND\_NXT} + 1) \bmod n\Big]\Big) \\
&\quad \lhd\ s = \mathit{ESTABLISHED} \wedge \mathit{length}(\mathit{get\_send\_buffer}(t)) = 0\ \rhd\ \delta \\
&+ \quad \mathit{tcp\_rcv\_CLOSE} \cdot \\
&\quad \mathit{tcp\_call\_SND}(\mathit{sgmt}(\mathit{get\_SND\_NXT}(t), \mathit{get\_RCV\_NXT}(t), \mathit{adv\_wnd}(t), 0, \mathtt{F}, \mathtt{T})) \cdot \\
&\quad \mathit{TCP}\Big(\mathit{LAST\_ACK}, t\Big[\mathit{rtq} \mapsto \mathit{add}(\mathit{sgmt}(\mathit{SND\_NXT}, \mathit{RCV\_NXT}, \mathit{adv\_wnd}(t), 0, \mathtt{F}, \mathtt{T}), \mathit{rtq}), \\
&\qquad\qquad \mathit{SND\_NXT} \mapsto (\mathit{SND\_NXT} + 1) \bmod n\Big]\Big) \\
&\quad \lhd\ s = \mathit{CLOSE\_WAIT} \wedge \mathit{length}(\mathit{get\_send\_buffer}(t)) = 0\ \rhd\ \delta
\end{aligned}
\tag{4.6}
$$

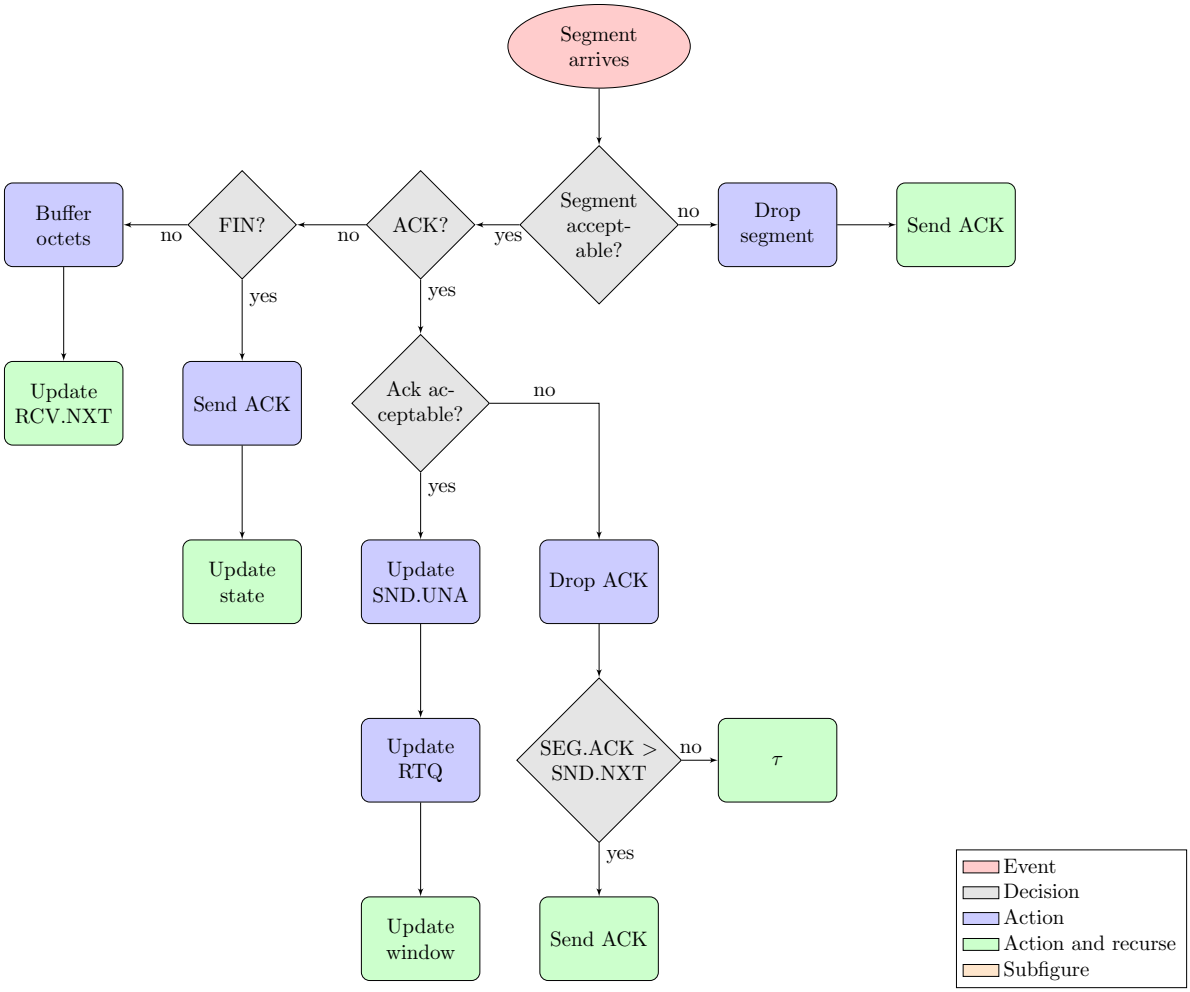


Figure 4.5: Abstract overview of our specification of TCP Segment Processing

Events

Apart from calls by the application layer, four events may occur that should trigger an action of the TCP instance: arrival of a segment, a user timeout, a retransmission timeout or a time-wait timeout. Of these events, a user timeout – which indicates that the user has not interacted with the connection for a specified period – is not included in our model as it does not add any behaviour except for deleting the connection.

The processing of an incoming segment is specified on pages 65-76 of [36]. For each segment, it is first checked whether it is acceptable through the function *receiver_may_accept*. The details of this check are discussed in detail in chapter 2. In our specification, however, the receive window is calculated from `RCV_WND`, `RCV_RD_NXT` and `RCV_NXT` through the function *rcv_wnd* as explained before, since we chose not to update `RCV.WND` for reasons discussed before.

After the acceptability check, all acceptable segments must be processed in the order as determined by the sequence numbers. The specification states that whenever a segment arrives out of order, it *may* be held on a separate queue to be processed as soon as the TCP instance has processed all earlier segments. However, we expect that including this behaviour will greatly increase the size of our state space and is not strictly required for the correctness of TCP. Therefore, we chose to drop segments that arrive out of order (`SEG_SEQ`$\neq$`RCV_NXT`) as well as segments that are not acceptable at all.

$$
\begin{aligned}
+ \quad \sum_{m:Sgmt} \Big(& tcp\_call\_RECV(m) \cdot tcp\_call\_SND(construct\_ack(t)) \cdot TCP(s,t) \\
& \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2, \\
& \quad CLOSING,\ LAST\_ACK,\ TIME\_WAIT\} \\
& \quad \wedge (\neg receiver\_may\_accept(m,t) \vee get\_seq\_nr(m) \neq get\_RCV\_NXT(t)) \ \rhd\ \delta \Big)
\end{aligned}
\tag{4.7}
$$

Note that whenever an unacceptable segment is received, an acknowledgement is sent immediately. Acknowledgements are constructed using the *construct_ack* function, that takes the current state of the TCB as a parameter. It then constructs an acknowledgement including the sequence number of the octet that the TCP instance is expected to receive next, the acknowledgement number and the advertised window. Obviously, the `ACK`-flag of the acknowledgement segment is set to `T` while the `FIN`-flag is set to `F`:

$$
construct\_ack(t) \quad = \quad sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), \mathtt{T}, \mathtt{F})
$$

Usually, a segment may contain control information, acknowledgement information and data. We already mentioned that in our model, a segment that carries data always has the `ACK` flag set to false. In addition, control segments in our model do not contain acknowledgement information or data. The only control information that we need to relay is information related to the closing of the connection. Hence, we distinguish between three types of segments:

1. A segment carrying data
2. A segment carrying acknowledgement information (an acknowledgement segment)
3. A segment carrying `FIN` information (a `FIN` segment)

By calling the function *is_acknowledgement* parameterised with the incoming segment, the TCP instance can determine whether it is an acknowledgement segment. Likewise, by calling the function *fin_flag_set*, the TCP instance can determine whether a segment is a `FIN` segment. If neither of these two is true, the segment is understood to be a segment carrying data. Processing is done for each of these situations separately.

If the incoming segment m is a `FIN` segment, it is processed as described on pages 75-76 of [36]. An acknowledgement is constructed and sent back to the remote end, after which the transmission control block is updated. Note that the state may progress from s to s': if the TCP instance was in state `ESTABLISHED`, `FIN_WAIT_1` or `FIN_WAIT_2` it progresses to state `CLOSE_WAIT`, `CLOSING` or `TIME_WAIT` respectively. In all other cases, the TCP instance stays in the same state. `RCV_ACK_QUEUED` is set to false since we immediately send an acknowledgement a for a `FIN` segment and any outstanding acknowledgement are included in a.

$$
\begin{aligned}
+ \quad & \sum_{m:Sgmt} \Big(tcp\_call\_RECV(m) \cdot \\
& tcp\_call\_SND(construct\_ack(t\Big[RCV\_NXT \mapsto (RCV\_NXT + 1) \bmod n\Big])) \cdot \\
& TCP\Bigg(s', t\Big[RCV\_NXT \mapsto (RCV\_NXT + 1) \bmod n, RCV\_ACK\_QUEUED \mapsto \mathtt{F}\Big]\Bigg) \\
& \lhd\ s \in \{ESTABLISHED,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2,\ CLOSE\_WAIT, \\
& \qquad CLOSING,\ LAST\_ACK,\ TIME\_WAIT\} \\
& \wedge receiver\_may\_accept(m,t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t) \\
& \wedge fin\_flag\_set(m)\ \rhd\ \delta \Big)
\end{aligned}
\tag{4.8}
$$

If the incoming segment m is an acknowledgement, the TCP instance must first verify whether the acknowledgement is acceptable. This is done by calling the function *may_accept_ack*. This function verifies whether $\mathtt{SND\_UNA} < get\_ack\_nr(m) \leq \mathtt{SND\_NXT}$. If this is the case, the acknowledgement is acceptable. However, it may be the case that the acknowledgement was delayed in the channel and that a newer acknowledgement, carrying the same acknowledgement number, has arrived in the meantime. Now, if the window information contained in this acknowledgement would be used to update the value of SND_WND, this could lead to degraded performance. Therefore, it is also decided whether the window information must be updated or not, by verifying whether or not $\mathtt{SND\_WL1} < get\_seq\_nr(m) \vee (\mathtt{SND\_WL1} = get\_seq\_nr(m) \wedge \mathtt{SND\_WL2} \leq get\_ack\_nr(m))$. This check is executed by the function *must_update_window*.

If the window must be updated, SND_WND is updated to $get\_window(m) * \mathtt{SND\_WND\_SCALE}$ and SND_WL1 and SND_WL2 are updated to the sequence and acknowledgement number of the segment respectively. In either case, SND_UNA is updated and all segments containing octets with a sequence number of at most i such that $\mathtt{SND\_UNA} \leq i < get\_ack\_nr(m)$ are removed from the retransmission queue. Also note that in the first two summands, our recursive call to TCP is parameterised with s' instead of s. The reason for this is that if the TCP instance is currently in state FIN_WAIT_1, CLOSING or LAST_ACK, the acknowledgement may acknowledge the FIN segment that the TCP instance has sent. If this is the case, the state is updated to FIN_WAIT_2, TIME_WAIT or CLOSED respectively. In all other cases, the TCP instance remains in the same state.

Finally, if the acknowledgement is not acceptable it is simply dropped and the TCP instance remains in the same state. Here, we again slightly deviate from the official specification, that states that in response to an unacceptable acknowledgement, an acknowledgement must be sent back if SEG.ACK > SND.NXT. However, we already discussed that in a unidirectional setting, such a situation will never occur and therefore, this behaviour is excluded from our model for the sake of simplicity but *must* be added if our specification is to be used in a bidirectional setting.

$$
\begin{aligned}
+ \quad & \sum_{m:Sgmt} \Big(tcp\_call\_RECV(m) \cdot \\
& TCP\Big(s', t\Big[rtq \mapsto update\_rtq(rtq, get\_ack\_nr(m), SND\_UNA), \\
& \qquad SND\_WL2 \mapsto get\_ack\_nr(m), SND\_WL1 \mapsto get\_seq\_nr(m), \\
& \qquad SND\_WND \mapsto get\_window(m) * SND\_WND\_SCALE, \\
& \qquad SND\_UNA \mapsto get\_ack\_nr(m) \Big] \Big) \\
& \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2, \\
& \qquad CLOSING,\ LAST\_ACK\} \\
& \wedge receiver\_may\_accept(m,t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t) \\
& \wedge is\_acknowledgement(m) \wedge may\_accept\_ack(m,t) \\
& \wedge must\_update\_window(m,t) \ \rhd\ \delta \Big) \\
+ \quad & \sum_{m:Sgmt} \Big(tcp\_call\_RECV(m) \cdot \\
& TCP\Big(s', t\Big[rtq \mapsto update\_rtq(rtq, get\_ack\_nr(m), SND\_UNA), \\
& \qquad SND\_UNA \mapsto get\_ack\_nr(m) \Big] \Big) \\
& \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2, \\
& \qquad CLOSING,\ LAST\_ACK\} \\
& \wedge receiver\_may\_accept(m,t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t) \\
& \wedge is\_acknowledgement(m) \wedge may\_accept\_ack(m,t) \\
& \wedge \neg must\_update\_window(m,t) \ \rhd\ \delta \Big) \\
+ \quad & \sum_{m:Sgmt} \Big(tcp\_call\_RECV(m) \cdot TCP(s,t) \\
& \lhd\ s \in \{ESTABLISHED,\ CLOSE\_WAIT,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2, \\
& \qquad CLOSING,\ LAST\_ACK\} \\
& \wedge receiver\_may\_accept(m,t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t) \\
& \wedge is\_acknowledgement(m) \wedge \neg may\_accept\_ack(m,t) \ \rhd\ \delta \Big)
\end{aligned}
\tag{4.9}
$$

If the incoming segment is an acceptable segment, for which it holds that (SEG_SEQ=RCV_NXT), and for which both *is_acknowledgement* and *fin_flag_set* return false, it is processed as a data segment. The octets in the segment are added to the receive buffer and RCV_NXT is updated to reflect the next sequence number that the receiver expects to receive. Furthermore, the RCV_ACK_QUEUED flag in the transmission control block, which indicates that an acknowledgement should be sent, is set to true:

$$
\begin{aligned}
+ \quad & \sum_{m:Sgmt} \Big(tcp\_call\_RECV(m) \cdot \\
& TCP\Big(s, t\Big[RCV\_NXT \mapsto (RCV\_NXT + get\_num\_octs(m)) \bmod n, \\
& \qquad receive\_buf \mapsto merge(receive\_buf, infl(get\_seq\_nr(m), get\_num\_octs(m))) \\
& \qquad RCV\_ACK\_QUEUED \mapsto \mathtt{T} \Big] \Big) \\
& \lhd\ s \in \{ESTABLISHED,\ FIN\_WAIT\_1,\ FIN\_WAIT\_2,\} \\
& \quad \wedge receiver\_may\_accept(m, tcb) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t) \\
& \quad \wedge \neg is\_acknowledgement(m) \wedge \neg fin\_flag\_set(m) \ \rhd\ \delta \Big)
\end{aligned}
\tag{4.10}
$$

In [36], it is stated that *"when the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data. [...] This acknowledgement should be piggybacked on a segment being transmitted if possible without incurring undue delay"*. [8] clarifies this point further, stating: *"a host [...] can increase efficiency in both the Internet and the hosts by sending fewer than one ACK (acknowledgement) segment per data segment received"*. While the delay that is referred to here is a performance enhancement, its impact is significant enough to justify an increase in the complexity of our model. Therefore, we include this behaviour in our model and do not let the TCP instance send an acknowledgement in the previous summand, but rather let it set a flag that an acknowledgement should be sent. We then include a separate summand that may send an acknowledgement whenever the RCV_ACK_QUEUED flag in the transmission control block is set to true. To prevent an acknowledgement from being sent multiple times, this flag is then set to false again. Note that acknowledgement segments are separated from data segments as we have not specified piggy-backing.

$$
\begin{aligned}
+ \quad & tcp\_call\_SND(construct\_ack(t)) \cdot TCP\Big(s, t\Big[RCV\_ACK\_QUEUED \mapsto \mathtt{F} \Big] \Big) \\
& \lhd\ get\_RCV\_ACK\_QUEUED(t) = \mathtt{T} \ \rhd\ \delta
\end{aligned}
\tag{4.11}
$$

This approach has the additional benefit that from a modelling perspective, it solves an ambiguity in both [36] and [8] about what window information must be included in the acknowledgement segment. By constructing and sending acknowledgements in a separate summand, the size of the receive buffer including the just received segments may be reflected (if the acknowledgement is sent immediately and no RECEIVE calls are processed in the meantime) or the size of the receive buffer without the just received segments (if the acknowledgement is only sent after all octets have been passed on to the application layer) and any possible situation in between these two extremes. We believe that this interpretation is closest to what was meant originally.

The second event that is discussed in [36] and included in our model is the retransmission timeout. For each segment that the TCP instance puts on the retransmission queue, it starts a timer. When a timer goes off, the corresponding segment must be retransmitted. To avoid modelling timing issues in our specification, we abstract away from this timer, and simply allow a TCP instance to retransmit the first

element on the retransmission queue - that has been on the retransmission queue for the longest time - at its own convenience at any time:

$$+ \quad tcp\_call\_SND(first(get\_rtq(t))) \cdot TCP(s,t) \ \lhd \ length(get\_rtq(t)) > 0 \ \rhd \ \delta \tag{4.12}$$

While this behaviour could have a negative impact on the performance of the protocol, we believe it does not significantly alter the behaviour compared to a situation in which timers are employed.

The final event that may occur is the time-wait timeout. A TCP connection may not transfer to the `CLOSED` state – a fictional state that in reality means that the connection no longer exists – before it is absolutely certain that the acknowledgement that it sent in response to a `FIN` segment is received at the other end. To this end, the connection must be kept alive for at least two times the maximum segment lifetime. Now, if the acknowledgement of the `FIN` segment gets lost, the remote end will eventually retransmit its `FIN` segment. If this segment arrives, the TCP entity will again respond with an acknowledgement and restart the time-wait timer. If eventually this timer goes off, the connection can be closed. As we did with the other timers in the specification, we abstract from this timer as well and include the following summand:

$$+ \quad tcp\_TW\_TIMEOUT \cdot TCP(CLOSED,t) \ \lhd \ s = TIME\_WAIT \ \rhd \ \delta \tag{4.13}$$

stating that if the TCP entity is in the `TIME_WAIT` state, it may progress to the `CLOSED` state. The `tcp_TW_TIMEOUT`-action is included since otherwise the recursion would be unguarded which is not allowed in μCRL.

Finally, we must add the following summand since μCRL cannot cope with terminating processes:

$$+ \quad tcp\_idle \cdot TCP(s,t) \ \lhd \ s = CLOSED \ \rhd \ \delta \tag{4.14}$$

stating that if the TCP entity is in the `CLOSED` state, it may perform the action `tcp_idle` and recurse. Later, we will ensure that the `tcp_idle` actions of both entities may only occur synchronised. Hence, even once both parties have successfully closed the connection there will be an action to be performed and as a result of this, the specification is not terminating.

4.3.3 The network layer

The final component that we must specify is the network layer. The goal of TCP is to enable a reliable communication between two parties through a network layer that may introduce faults into the communication. Hence, we specify a faulty network layer that may duplicate, reorder and lose data. Since TCP specifically mentions the IP protocol as one of possible implementations for the network layer, we adopt the naming conventions as used in [35]:

$$
\begin{aligned}
&NL(q{:}SgmtQueue) = \\
&\quad \sum_{m:Sgmt} \Big(nl\_rcv\_SEND(m) \cdot NL(add\_ordered(m,q)) \\
&\quad \lhd\ length(q) < medium\_capacity\ \rhd\ \delta \Big) \\
&+ \quad \sum_{m:Sgmt} \Big(nl\_rcv\_RECV(m) \cdot NL(remove(m,q)) \\
&\quad \lhd\ get\_seq\_nr(m) < n \wedge m \in q\ \rhd\ \delta \Big) \\
&+ \quad \sum_{m:Sgmt} \Big(nl\_rcv\_SEND(m) \cdot NL(q) \\
&\quad \lhd\ length(q) < medium\_capacity\ \rhd\ \delta \Big) \\
&+ \quad \sum_{m:Sgmt} \Big(nl\_rcv\_RECV(m) \cdot NL(q) \\
&\quad \lhd\ get\_seq\_nr(m) < n \wedge m \in q\ \rhd\ \delta \Big) \\
&+ \quad medium\_drain \cdot NL(rest(q))\ \lhd\ length(q) = medium\_capacity\ \rhd\ \delta
\end{aligned}
\tag{4.15}
$$

The first two summands represent a regular network layer. First of all, by issuing a `nl_rcv_SEND` call, the network layer accepts an octet and puts it in its buffer. To prevent state space explosion, this buffer is implemented as a list of segments ordered by their sequence number. By issuing a `nl_rcv_RECV` call, an arbitrary element in the buffer is removed and put on the output channel. At the same time, this introduces the first of the three possible faults; reordering.

Figure 4.6: Conceptual overview of the Network Layer and its channels

The second two summands represent the two other possible faults: losing and duplicating a segment. Losing is specified as accepting a segment by issuing a `nl_rcv_SEND` call, but not putting the accepted segment in the buffer. Duplicating is specified as putting a segment on the output channel by issuing a `nl_rcv_RECV` call but not removing the segment from the buffer.

Finally, the fifth summand is included to prevent the buffer that the network layer maintains from becoming full, as this may result in a deadlock. It simply removes the first element from the buffer. While this behaviour is not explicitly specified in the RFCs, it mimics TCP's maximum segment lifetime when necessary and is therefore in accordance with a real-world implementation.

4.3.4 The complete system

We obtain the complete system, as shown in figure 4.1, by putting the components as described above in parallel. The renaming operator ρ is used to rename general action names into action names specific for each component. We assume the following variables to be set as a result of the connection establishment procedure:

- *initial_seq_num_TCP1*, *initial_seq_num_TCP2*
  The initial sequence number that the TCP instances will use to send a data segment.

- *initial_window_size_TCP1*, *initial_window_size_TCP2*
  The initial size of the send window of each of the TCP instances.

- *window_scale_TCP1*, *window_scale_TCP2*
  The scale factor that each of the TCP instances will apply to their outgoing segments. Note that both TCP instances know of each other's scale factors, since they need to reversely apply the scale factor to incoming segments. This is arranged for during connection establishment as discussed in chapter 2.
  Our use of the scale factor is slightly different from the use of the scale factor in [23], where the factor is defined as n, resulting in integer division/multiplication by 2^n as a result of bit shifting. Throughout this thesis, the scale factor is maintained in as a value of 2^n since we use division and multiplication operations to perform the scaling. Hence, if a scale factor of 1 as mentioned in [23] is meant, we will write a scale factor of $2 = 2^1$.

- *total_octets_to_send_TCP1*, *total_octets_to_send_TCP2*
  The number of octets that each TCP instance should send.

Furthermore, the following global variables are assumed to be defined:

- n
  The total number of sequence numbers. Used in modulo calculations and for the upper bounds in the specification of the medium.
  When choosing a value for n, we ensure that $n \geq max(window\_size\_TCP1, window\_size\_TCP2) * 2$.

- *medium_capacity*
  The number of segments that the medium can hold at most.

- *buffer_capacity*
  The number of octets that the send buffer can hold at most. When choosing a value for *buffer_capacity*, we ensure that *buffer_capacity*$\geq max(window\_size\_TCP1, window\_size\_TCP2)$

The specification of the complete system is shown in figure 4.7.

Summary

In this chapter, we gave a detailed overview of our specification of TCP. We started by defining the scope of our specification, and explained which features were left out of it. After an explanation of the data types that we defined, we discussed the specification of each component in detail. Finally, we showed how to compose these components to obtain our *SystemSpecification*.

In the next chapter, we will show how to obtain a model from this *SystemSpecification* and discuss our verification approach and the results that our verification yielded.

$$
\begin{aligned}
&\mathit{SystemSpecification} = \Big(\\
&\quad \rho_{\{al\_call\_SEND \to AL1\_call\_SEND, al\_call\_RECEIVE \to AL1\_call\_RECEIVE,} \\
&\qquad al\_call\_CLOSE \to AL1\_call\_CLOSE\}\big(\\
&\qquad AL(initial\_seq\_num\_TCP1, total\_octets\_to\_send\_TCP1)\big) \parallel \\
&\quad \rho_{\{tcp\_rcv\_SND \to TCP1\_rcv\_SEND, tcp\_rcv\_RECEIVE \to TCP1\_rcv\_RECEIVE,} \\
&\qquad tcp\_call\_SND \to TCP1\_call\_SEND, tcp\_call\_RECV \to TCP1\_call\_RECV, \\
&\qquad tcp\_TW\_TIMEOUT \to TCP1\_TW\_TIMEOUT, tcp\_idle \to TCP1\_idle, \\
&\qquad tcp\_rcv\_CLOSE \to TCP1\_rcv\_CLOSE\}\big(\\
&\qquad TCP(ESTABLISHED,\ tcb(\emptyset,\ \emptyset,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP1, \\
&\qquad initial\_window\_size\_TCP2,\ 0,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP1, \\
&\qquad window\_scale\_TCP2,\ \emptyset,\ initial\_seq\_num\_TCP2,\ initial\_window\_size\_TCP1,\ 0,\ initial\_seq\_num\_TCP2, \\
&\qquad window\_scale\_TCP1,\ initial\_seq\_num\_TCP2, \mathtt{F}))) \parallel \\
&\quad \rho_{\{nl\_rcv\_SEND \to NL1\_rcv\_SEND, nl\_rcv\_RECV \to NL1\_rcv\_RECV\}}\big(NL(\emptyset)\big) \parallel \\
&\quad \rho_{\{nl\_rcv\_SEND \to NL2\_rcv\_SEND, nl\_rcv\_RECV \to NL2\_rcv\_RECV\}}\big(NL(\emptyset)\big) \parallel \\
&\quad \rho_{\{tcp\_rcv\_SND \to TCP2\_rcv\_SEND, tcp\_rcv\_RECEIVE \to TCP2\_rcv\_RECEIVE,} \\
&\qquad tcp\_call\_SND \to TCP2\_call\_SEND, tcp\_call\_RECV \to TCP2\_call\_RECV, \\
&\qquad tcp\_TW\_TIMEOUT \to TCP2\_TW\_TIMEOUT, tcp\_idle \to TCP2\_idle, \\
&\qquad tcp\_rcv\_CLOSE \to TCP2\_rcv\_CLOSE\}\big(\\
&\qquad TCP(ESTABLISHED,\ tcb(\emptyset,\ \emptyset,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP2, \\
&\qquad initial\_window\_size\_TCP1,\ 0,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP2, \\
&\qquad window\_scale\_TCP1,\ \emptyset,\ initial\_seq\_num\_TCP1,\ initial\_window\_size\_TCP2,\ 0,\ initial\_seq\_num\_TCP1, \\
&\qquad window\_scale\_TCP2,\ initial\_seq\_num\_TCP1, \mathtt{F}))) \parallel \\
&\quad \rho_{\{al\_call\_SEND \to AL2\_call\_SEND, al\_call\_RECEIVE \to AL2\_call\_RECEIVE,} \\
&\qquad al\_call\_CLOSE \to AL2\_call\_CLOSE\}\big(\\
&\qquad AL(initial\_seq\_num\_TCP2, total\_octets\_to\_send\_TCP2)\big) \\
&\Big)
\end{aligned}
$$

Figure 4.7: Our *SystemSpecification*, obtained by putting our components together

Chapter 5

Formal verification of TCP

5.1 Introduction

In the previous chapter, we described our specification of TCP data transfer and connection teardown in detail. To perform a verification, our *SystemSpecification* is first transformed into a *model*, by parameterising it with variables such as the initial sequence number, window size, window scale factor etc. This model is subsequently used to generate a *state space*. With this state space, the correctness of the model can be verified using either of the approaches as discussed in chapter 3.

However, when we verified earlier versions of our model we soon found out that state space generation for the complete model was going to be an infeasible task even for unidirectional TCP. It appeared that the connection teardown procedure has an enormous impact on the size of the state space, since TCP connections are terminated in a simplex fashion. Hence, while the sending TCP entity can only close its connection after sending n octets, the receiving TCP may close its connection at any time, causing the entire connection termination procedure to be included after almost every transition from one state to another in our model.

We thought of several possibilities to alleviate this complexity, but none of them led to satisfactory results. Our first thought was to have the receiving entity only close the connection after it received all of the n segments that the sending entity sent. This, however, would deviate so much from real-world behaviour that we concluded it to be an uninteresting situation to study. Another thought was to initialise the receiving entity in the `CLOSED` state. However, this would completely ignore half of the connection termination procedure – and we think all interesting situations that may occur – and could therefore not be considered as a verification of connection teardown.

Besides issues with the size of the state space, we ran into some undesired behaviour and even deadlock scenarios that were a result of the reuse of sequence numbers and absence of a timing mechanism to enforce the Maximum Segment Lifetime (MSL) as assumed in [36].

To overcome the first problem, we decided to split our verification into two distinct parts. First of all, we performed a verification of unidirectional TCP without connection termination. Second, we performed another verification that only considers the connection teardown procedure. To tailor our *SystemSpecification* to each of these approaches and to prevent unwanted behaviour and deadlock scenarios from occurring, some small modifications had to be made.

In this chapter, for each verification we will discuss these modifications after explaining the issues that forced us to make them. Then, we will show how to obtain the model of interest from this modified specification and give the details of our verification procedure. Finally, we will discuss the results of the verification procedure and their implications.

5.2 Formal verification of $TCP_{\rightarrow}$

5.2.1 A model of $TCP_{\rightarrow}$

From the start of our project, we took an incremental approach to our modelling. When we started generating state spaces for early versions of our model, that did not yet include many features of TCP, we already found that state spaces were becoming too large in situations where network traffic was bidirectional. Therefore, we focused our work on unidirectional network traffic, where data segments flow in one direction and acknowledgements in the other.

In general, to obtain a model from the *SystemSpecification*, we must specify three sets. The first of these sets is Θ and contains pairs of actions that synchronise. The second set, τ, contains those actions that represent the internal actions. In the state space that is generated for the model, all transitions that represent an action $a \in \tau$ will be labelled with τ instead of the action name a. Finally, the set δ contains all actions that are included in the *SystemSpecification* but that may not occur in our model. In this set, we include all action names of the separate components that are also contained in Θ to ensure that these actions – which are meant to be synchronising – will only occur in a synchronised fashion.

More specifically, in order to obtain a model for unidirectional TCP, the actions `AL2_call_SEND` and `TCP2_rcv_SEND` were encapsulated by including them in δ. At the same time, they were left out of Θ to prevent AL2 from issuing the `SEND` call, while the actions `AL1_call_RECEIVE` and `TCP1_rcv_RECEIVE` were included in δ and left out of Θ to prevent AL1 from issuing the `RECEIVE` call. Hence, AL1 only issues the `SEND` call to pass octets to TCP1. TCP1 will subsequently transmit these octets over NL1 to TCP2, which will send its acknowledgements back over NL2. Finally, AL2 may issue `RECEIVE` calls to retrieve the octets in the receive buffer of TCP2.

Additionally, we had to exclude connection termination from our model to make generation of the state space feasible. The approach is similar to the approach as described above: by encapsulating the actions `AL1_call_CLOSE`, `AL2_call_CLOSE`, `TCP1_rcv_CLOSE` and `TCP2_rcv_CLOSE` and excluding them from Θ, we ensured that they will not be called in our model. However, as a result of this, both TCP instances will never reach the `CLOSED` state. Therefore, we also had to adapt summand 4.14 to prevent our model from terminating. It was replaced by the following summand:

$$
\begin{aligned}
+ \quad & \mathit{tcp\_idle} \cdot \mathit{TCP}(s,t) \\
& \lhd\ \mathit{length}(\mathit{get\_send\_buffer}(t)) = 0 \wedge \mathit{length}(\mathit{get\_receive\_buf}(t)) = 0 \\
& \quad \wedge \mathit{length}(\mathit{get\_rtq}(t)) = 0\ \rhd\ \delta
\end{aligned}
\tag{5.1}
$$

Taken together, these modifications ensured that we were able to obtain a non-terminating model $TCP_{\rightarrow}$ of the data-transfer phase for a unidirectional TCP connection from our *SystemSpecification*. Figure 5.1 shows an overview of the sets Θ, τ and δ.

Finally, we had to prevent the deadlock scenarios and undesired behaviour as a result of the reuse of sequence numbers from occurring in the state space as generated from our model. In chapter 2, these issues were already discussed in detail and we showed that, while the introduction of PAWS as proposed in [23] helps to alleviate this problem when networks get faster (by increasing the size of the sequence number space) no solid protection mechanism has been proposed so far, other than the assumption that segments that are in the medium have a maximum lifetime.

We have also shown that if we have 2^n sequence numbers, the maximum window size that may be used is 2^{n-2} in order to prevent deadlock scenarios or unwanted behaviour. As can be seen from the overview that table 5.1 shows, state space generation with a sequence number space of size 2^4 turned out to be infeasible. As a result of this, the maximum size of the sequence number space is 2^3 and from the specification as highlighted in chapter 2, it follows that the maximum window size that may be used with 2^3 sequence numbers is 2^1. This, however, conflicts with the fact that for the calculations made for the window scaling option to be non-trivial, we believe that the window size should be at least 2^2 for

reasons we will discuss later in this chapter. Hence, even if we would add a timing mechanism to our specification that enforces the MSL, problems with the reuse of sequence numbers remain and we must find another solution.

The nature of the assumption on the maximum lifetime of segments is that TCP must not reuse a sequence number until it is absolutely sure that no segment with (an acknowledgement of) that sequence number is in the medium. Strictly speaking, this means that TCP cycles through all of its sequence numbers, waits until all segments have been acknowledged and all duplicates have drained from the network and starts a new run. The only reason that in practice there is no actual waiting period, is that data transfer speeds are guaranteed to be so 'slow' that it takes more than the MSL to send out octets with sequence numbers $i + 1 \ldots (i + (n - 1)) \bmod n$ after an octet with sequence number i is sent.

For this reason, we think that it is reasonable to limit our verification to one run of sequence numbers. This has the benefit that our specification does not get overly complex due to the addition of timing restrictions or other synchronisations between the TCP instances and the mediums to ensure that all segments with a sequence number m drain from the network before the TCP instance reuses it. Also, note that while our specification does not include the reuse of sequence numbers, we may still start anywhere in the sequence number space, since all calculations are defined modulo the size of the sequence number space.

However, this did not solve all of our problems, as the following scenario shows. Assume that we have a sequence number space ranging from $m \ldots n$. The receiving TCP instance will still accept an octet with sequence number m after it has received the octet with sequence number n. Hence, if such an octet is still in the medium as the result of duplication or retransmission and delivered to the receiving TCP instance, it will be accepted and subsequently delivered to the application layer. Naturally, such behaviour is undesired even in our model, since it breaks bisimilarity with a FIFO Queue in which octets are only accepted once, a property that we want to prove. More importantly, given that the assumption on the MSL holds, it can not occur in a real-world situation.

We propose the following fix for this problem: if we ensure that the value assigned to the global variable that maintains the total number of sequence numbers is greater than the number of octets, the sequence number space is guaranteed to be larger than the number of octets that is sent and the problem will not occur. To see why this is the case, assume that we want to send m octets and that we have a sequence number space of size $n = m + 1$. Now, after receiving octets $i \ldots (i + m) \bmod n$, the receiving TCP instance will expect an octet with sequence number $m + 1$ rather than i. As a result of this, delivery of a segment with sequence number i, which may still be in the medium as the result of duplication or retransmission, will be correctly seen as an unacceptable segment and dropped.

5.2.2 Verification

After having obtained a model from our *SystemSpecification*, we could verify the correctness of $TCP_{\rightarrow}$. First, we will define what correctness entails. In our research question, we formulated the following hypothesis for incorrect behaviour of the protocol:

1. There is a byte that is delivered to the receiver's transport layer corrupted, or
2. There are bytes that are delivered to the application layer of the receiver in a different order as the order in which they were delivered to the transport layer of the sender, or
3. There are bytes that are delivered to the application layer of the sender, but never delivered to the application layer of the receiver, or
4. There are scenarios in which the execution of the protocol gets stuck

For reasons discussed in chapter 4, our specification abstracts from corrupted segments. Furthermore, we have seen that TCP performs a check on each incoming segment to see whether it starts with the octet that it expects to receive next. If this is not the case, the segment is dropped. If this check is implemented correctly, it is not possible that bytes are delivered to the application layer of the receiver

$$
\begin{aligned}
TCP_{\rightarrow} = \Big(\\
&\Theta_{\{AL1\_call\_SEND | TCP1\_rcv\_SEND \rightarrow SEND} \\
&\quad TCP1\_call\_SEND | NL1\_rcv\_SEND \rightarrow TCP1\_to\_NL1 \\
&\quad TCP1\_call\_RECV | NL2\_rcv\_RECV \rightarrow TCP1\_from\_NL2 \\
&\quad TCP2\_call\_SEND | NL2\_rcv\_SEND \rightarrow TCP2\_to\_NL2 \\
&\quad TCP2\_call\_RECV | NL1\_rcv\_RECV \rightarrow TCP2\_from\_NL1 \\
&\quad TCP1\_idle | TCP2\_idle \rightarrow CONNECTION\_CLOSED \\
&\quad AL2\_call\_RECEIVE | TCP2\_rcv\_RECEIVE \rightarrow RECEIVE\} \Big(\\
&\quad \tau_{\{TCP1\_to\_NL1,\ TCP1\_from\_NL2,\ TCP2\_from\_NL1,\ TCP2\_to\_NL2,\ CONNECTION\_CLOSED\}} \Big(\\
&\qquad \delta_{\{AL1\_call\_SEND,\ AL1\_call\_RECEIVE,\ AL1\_call\_CLOSE,} \\
&\qquad\quad TCP1\_rcv\_SEND,\ TCP1\_rcv\_RECEIVE,\ TCP1\_call\_SEND,\ TCP1\_call\_RECV,\ TCP1\_rcv\_CLOSE, \\
&\qquad\quad AL2\_call\_SEND,\ AL2\_call\_RECEIVE,\ AL2\_call\_CLOSE, \\
&\qquad\quad TCP2\_rcv\_SEND,\ TCP2\_rcv\_RECEIVE,\ TCP2\_call\_SEND,\ TCP2\_call\_RECV,\ TCP2\_rcv\_CLOSE, \\
&\qquad\quad NL1\_rcv\_SEND,\ NL1\_rcv\_RECV,\ NL2\_rcv\_SEND,\ NL2\_rcv\_RECV,\ TCP1\_idle,\ TCP2\_idle\} \Big(\\
&\qquad\quad SystemSpecification \\
&\qquad \Big) \\
&\quad \Big) \\
&\Big)
\end{aligned}
$$

Figure 5.1: Obtaining a model for $TCP_{\rightarrow}$ from our *SystemSpecification*

in a different order as the order in which they were delivered to the transport layer of the sender.[10] The third and fourth case, however, are more likely to occur in the event of an error.

Therefore, our verification efforts focused on two aspects of our model. First of all, we verified that its state space does not contain any deadlocks, since they are a sign of an error as a result of how we approached the specification. If m octets must be sent and everything works correctly, the application layer at the sending end issues the `SEND` call m times and the receiving TCP instance issues the `RECEIVE` call m times, after which the protocol will indefinitely perform the `CONNECTION_CLOSED` action. It should be clear that a deadlock does not fit into this expected behaviour, and that therefore deadlocks may not occur in our model.

Second, we compared the external behaviour of our model, defined in terms of the `SEND` and `RECEIVE` calls as issued by the application layers at either end, to the external behaviour of a FIFO queue. We chose this approach over model checking since we expected the latter technique to be infeasible on a state space as large as that of $TCP_{\rightarrow}$. The reason to model the external behaviour as a FIFO queue is that from a such a queue, elements are taken in the same order as that they were put into the queue. If the `SEND` call of TCP is seen as putting something into a queue and the `RECEIVE` call as taking something from it, it becomes immediately clear that the external behaviour of the FIFO queue is the desired behaviour for TCP: the sending entity puts some data elements into the transport medium and the receiving entity takes them all out of this medium in exactly the same order.

In order to perform this comparison, we first specified a behavioural specification B. Then, we generated an LTS for B and $TCP_{\rightarrow}$. Recall that in $TCP_{\rightarrow}$, all actions other than `SEND` and `RECEIVE` were defined as internal behaviour by including them in the set τ. We minimised the LTS of $TCP_{\rightarrow}$ and then compared it to the LTS of B to verify that they are branching bisimilar: $TCP_{\rightarrow} \leftrightarrows_B B$. Note that, as discussed in chapter 3, a fairness assumption is enforced by the minimisation algorithm and therefore, τ-loops from which an 'exit' is possible were eliminated from our minimised state space. Examples of such τ-loops are segments that are continuously dropped by the network layer or a sequence of repeated retransmissions, behaviour that we can safely abstract from. In addition, by the fact that our behavioural specification does not contain livelocks, absence of livelocks in the external behaviour is guaranteed by showing $TCP_{\rightarrow} \leftrightarrows_B B$.

The behavioural specification

An abstract view of $TCP_{\rightarrow}$ is that the protocol consists of a sender S and a receiver R that communicate over a medium M. S reads octets from an application layer through the execution of an action `SEND` and puts these octets in a buffer. The protocol is then responsible for the transfer of messages to R over M, where they are again put into a buffer. The application layer of R may then consume the octets through the execution of an action `RECEIVE`. If the protocol functions correctly, the output generated by R is a prefix of the input consumed by S at any moment during the execution of the protocol.

Our behavioural specification follows from this immediately. Figure 5.2 shows a conceptual overview. It consists of three FIFO queues, one for the buffer located at the sender, one for the buffer located at the receiver and one that represents the medium. These queues are put in sequence, to represent the flow of messages from the sender's buffer to the receiver's buffer through the medium.

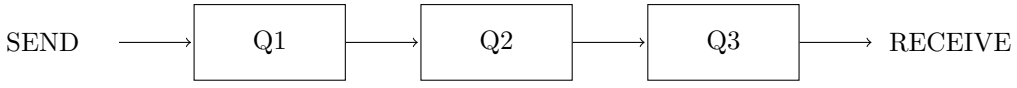


Figure 5.2: Conceptual overview of our behavioural specification

We first specify an additional data type *FIFOBuf*, representing a FIFO buffer of natural numbers. $\emptyset$ and qu are defined as constructors for the empty buffer and a natural number in the buffer respectively and several operations on the buffer are defined:

[10] Since they are not accepted by the receiving TCP instance to begin with, they can never be delivered to the application layer by that same TCP instance.

$$
\begin{array}{lcl}
\emptyset & : & \rightarrow \mathit{FIFOBuf} \\
\mathit{qu} & : & \mathit{Nat} \times \mathit{FIFOBuf} \rightarrow \mathit{FIFOBuf} \\
= & : & \mathit{FIFOBuf} \times \mathit{FIFOBuf} \rightarrow \mathit{Bool} \\
\mathit{first} & : & \mathit{FIFOBuf} \rightarrow \mathit{Nat} \\
\mathit{rest} & : & \mathit{FIFOBuf} \rightarrow \mathit{FIFOBuf} \\
\mathit{add} & : & \mathit{Nat} \times \mathit{FIFOBuf} \rightarrow \mathit{FIFOBuf} \\
\mathit{length} & : & \mathit{FIFOBuf} \rightarrow \mathit{Nat}
\end{array}
\qquad
\begin{array}{lcl}
\emptyset = \emptyset & = & \mathtt{T} \\
\mathit{qu}(x, q_1) = \mathit{qu}(y, q_2) & = & x = y \wedge q_1 = q_2 \\
\mathit{first}(\mathit{qu}(x, q_1)) & = & x \\
\mathit{rest}(\emptyset) & = & \emptyset \\
\mathit{rest}(\mathit{qu}(x, q_1)) & = & q_1 \\
\mathit{add}(x, \emptyset) & = & \mathit{qu}(x, \emptyset) \\
\mathit{add}(x, \mathit{qu}(y, q_1)) & = & \mathit{qu}(y, \mathit{add}(x, q_1)) \\
\mathit{length}(\emptyset) & = & 0 \\
\mathit{length}(\mathit{qu}(x, q_1)) & = & 1 + \mathit{length}(q_1)
\end{array}
$$

The FIFO queue is then specified as follows:

$$
\begin{array}{l}
\mathit{Queue}(x{:}\mathit{Nat}, \mathit{capacity}{:}\mathit{Nat}, q{:}\mathit{FIFOBuf}, y{:}\mathit{Nat}) = \\
\qquad \mathit{queue\_in}(x) \cdot \mathit{Queue}((x+1) \bmod n, \mathit{capacity}, \mathit{add}(x, q), y-1) \ \lhd\ y > 0 \wedge \mathit{length}(q) < \mathit{capacity} \ \rhd\ \delta \\
+ \quad \mathit{queue\_out}(\mathit{first}(q)) \cdot \mathit{Queue}(x, \mathit{capacity}, \mathit{rest}(q), y) \ \lhd\ \mathit{length}(q) > 0 \ \rhd\ \delta \\
+ \quad \mathit{idle} \cdot \mathit{Queue}(x, \mathit{capacity}, q, y) \ \lhd\ y = 0 \wedge \mathit{length}(q) = 0 \ \rhd\ \delta
\end{array}
$$

It is initialised with a natural number m, that represents the number that it expects to receive, a capacity, an ordered buffer of natural numbers and a natural number n that represents the total number of elements that must be put on the queue. As long as $n > 0$, whenever it receives a number m over its input channel `queue_in`, it puts it at the back of its buffer and progresses to wait for a number $m + 1$. In addition, whenever there are elements in the buffer, it may put the first one in the buffer onto its output channel `queue_out`. After it has added and subsequently removed n numbers from its buffer, it continuously executes the `idle` action to prevent the process from successfully terminating.

We obtained our behavioural specification by putting three of these queues in sequence. All three queues have a capacity that equals the window size n. Note that we used the variables *total_octets_to_send*, *initial_seq_num_TCP1* and *window_size_TCP1* as defined in our *SystemSpecification* and renamed the `queue_in` action of Q1 to `SEND` and the `queue_out` action of Q3 to `RECEIVE`. Figure 5.3 shows an overview.

State space generation

To obtain a state space, we first generated an LPO from our μCRL specification and subsequently converted this LPO to the LPS format of the `mCRL2` toolset. The reason that we converted from an LPO to an LPS is that during the course of our project, we regularly needed the `mCRL2` toolset to find errors in our specification. The `mCRL2` toolset turned out to be more useful to diagnose such errors as, contrary to the `ltsmin` toolset, it includes tools to simulate and visualise the state space, as well as tools that report counter-examples for branching bisimilarity. While we initially also specified the protocol in `mCRL2`, we ran into some issues that forced us to fall back on μCRL.

For both $TCP_{\rightarrow}$ and B, we generated a state space using the distributed state space generation tool `lps2lts-dist`[11] of the ltsmin toolset. By using the `--deadlock` option, absence of deadlocks could be checked during state space generation. In addition, we used the `--cache` option to speed up state space generation. State space generation was run on the DAS-4 cluster, more specifically on 8 nodes equipped with an Intel Sandy Bridge E5-2620 processor clocked at 2.0 GHz, 64 gigabytes of memory and a K20m Kepler GPU with 6 gigabytes of on-board memory. At each node, we utilised only 1 core to prevent the process from running out of memory. Table 5.1 shows some benchmarks of the state space generation for $TCP_{\rightarrow}$ for several different parameterisations.

[11] We also used lpo2lts-dist to generate a state space directly from the LPO obtained from the μCRL specification. The benchmarks are obtained with lps2lts-dist.

$$
\begin{aligned}
&\Theta_{\{sender\_queue\_out|medium\_queue\_in \rightarrow internal\_to\_medium} \\
&\quad medium\_queue\_out|receiver\_queue\_in \rightarrow internal\_from\_medium\} \Big(\\
&\quad \tau_{\{internal\_to\_medium,\ internal\_from\_medium,\ idle\}} \Big(\\
&\quad\quad \delta_{\{sender\_queue\_out,\ receiver\_queue\_in,\ medium\_queue\_in,\ medium\_queue\_out\}} \Big(\\
&\quad\quad\quad \rho_{\{queue\_in \rightarrow SEND, queue\_out \rightarrow sender\_queue\_out\}} (\\
&\quad\quad Queue(initial\_seq\_num\_TCP1, window\_size\_TCP1, \emptyset, total\_octets\_to\_send)) \parallel \\
&\quad\quad\quad \rho_{\{queue\_in \rightarrow medium\_queue\_in, queue\_out \rightarrow medium\_queue\_out\}} (\\
&\quad\quad Queue(initial\_seq\_num\_TCP1, window\_size\_TCP1, \emptyset, total\_octets\_to\_send)) \parallel \\
&\quad\quad\quad \rho_{\{queue\_in \rightarrow receiver\_queue\_in, queue\_out \rightarrow RECEIVE\}} (\\
&\quad\quad Queue(initial\_seq\_num\_TCP1, window\_size\_TCP1, \emptyset, total\_octets\_to\_send)) \parallel \\
&\quad\quad \Big) \\
&\quad \Big) \\
&\Big)
\end{aligned}
$$

Figure 5.3: Obtaining a model for our behavioural specification

Octets sent	Window size	Window scale	Medium capacity	Levels	States	Transitions	Exploration time	Completed
4	2	1	2	36	881.043	3.910.863	21 seconds	√
			3	40	11.490.716	53.137.488	104 seconds	√
			4	44	91.821.900	434.372.541	7.5 minutes	√
8	2	1	2	54	16.126.380	76.356.475	3 minutes	√
			3	58	823.501.590	4.031.264.559	49 minutes	√
	4	2	2	49	98.697.902	473.332.511	15 minutes	√
			3	56	3.505.654.685	Unknown[1]	3 hours, 40 minutes	√
16	4	2	2	77	3.255.174.492	3.444.088.224	4 hours, 40 minutes	√

Table 5.1: Statistics of the state space generation for our model

Subsequently, the state space of $TCP_{\rightarrow}$ was minimised with the `lts-reduce-dist` tool of the `ltsmin` toolset, which uses the distributed minimisation algorithm as described in [7]. Finally, the `ltsmin-compare` tool of the `ltsmin` toolset was used to verify that $TCP_{\rightarrow} \leftrightarrows_B B$.

5.2.3 Results

As can be seen from benchmarks in table 5.1, the capacity of the medium has a significant impact on the size of the state space. Therefore, we settled for a medium capacity of two segments. Since one segment may contain at most as many segments as the size of the window, a medium capacity of 2 means that a TCP instance can send two windows worth of data segments into the medium before it must 'wait' for the medium to either lose or deliver a segment.

For the window scaling to be non-trivial, we believe that the size of the window should be at least 2^2 with a scale factor of 2^1. If a window size of 2^1 with a scale factor of 2^1 would be chosen, the size of the window is either two or zero, meaning that the sending instance can either send its entire window at once, or nothing at all. By choosing 2^2 for the window size and a scale factor of 2^1, we have three possible window sizes: zero, two and four that allow for interesting scenarios where the reported size of the window is shrunken to half of its original size.

Hence, we performed our verification as discussed above for a model with a sequence number space of size 2^3+1, a window size of 2^2, a scale factor of 2^1 and a medium capacity of 2^1 segments, in which the sending TCP sends 2^3 octets. With these parameters, we obtained a model that was small enough to verify within reasonable time, with characteristics that are representative for a real-world implementation of TCP. We obtained models $TCP_{\rightarrow m}$ for initial sequence numbers $m \in \{0 \ldots 2^3\}$ and successfully verified that for each model $TCP_{\rightarrow m} \leftrightarrows_B B$.

While the model that we obtained with these parameters is relatively small, we believe that it is enough given the general nature of the specification. If the size of the model increases, all relevant buffers and calculations will simply scale with this increase. Therefore, we think that it is highly unlikely that errors are introduced as a result of increasing the size of the model.

[1] Due to a buffer overflow in the counter

Validity of our verification approach

To show that our verification approach correctly identifies the errors that we discussed in section 5.2.2, we will give two examples of an incorrect specification, in which we deliberately introduce either of the errors, and show that our verification approach identifies these specifications as incorrect. In the first example, we remove a TCP feature from the specification that is crucial to recover from errors that may occur in the medium, while in the second example we alter the behaviour of the receiving TCP instance such that it does not accept the last octet.

The TCP retransmission mechanism is crucial for correct operation of TCP over a lossy network: if a segment gets lost, it must be retransmitted. Otherwise, the segment will never arrive at the receiver and the byte stream will be corrupted. Therefore, we remove summand 4.12 from our specification and generate a state space for a model with a window size of 2^1, a scale factor of 2^0 a medium capacity of 2^1 segments and initial sequence number 1, and let the sending TCP send 2^2 octets. The first deadlock scenario that we find is shown in figure 5.4.

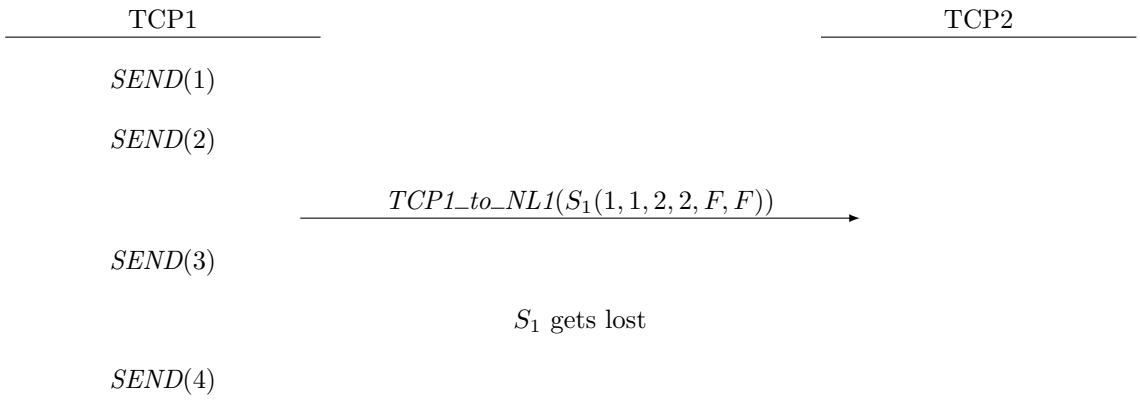


Figure 5.4: Deadlock scenario if a TCP entity does not retransmit segments

In this scenario, we see that the application layer at TCP1 issues the `SEND` call two times, for octets with sequence numbers 1 and 2. Subsequently, it sends a segment S_1 into the medium with sequence number 1 that carries two octets. Note that the segment also includes acknowledgement number 1, a reported window of size 2 and that the `ACK` and `FIN` flags are set to false. After sending this segment, the `SEND` call is issued twice for octets 3 and 4. Meanwhile, the segment that was sent into the medium gets lost.

The reason that the protocol now deadlocks is as follows. TCP1 has sent a segment carrying two octets and has updated `SND_NXT` to $1+2=3$. As a result of this, it can no longer send a segment since `SND_UNA` + `SND_WND` $= 1+2=3=$ `SND_NXT`. After filling the buffer to its maximum capacity, there is nothing left for TCP1 to do. TCP2 on the other hand never receives something and therefore no action is required.

For our second example, we alter the behaviour of TCP's `RECEIVE` call such that this call is not issued for the octet with sequence number 4, by adapting summand 4.5. Furthermore, summand 5.1 is adapted such that the `tcp_idle` call may also be made if the length of the receive buffer equals one and the octet on the receive buffer has sequence number 4. Now, state space generation will succeed and no deadlocks will be found. However, the resulting state space is not branching bisimilar with that of a FIFO Queue.

Correctness of the Window Scale Option

When we started this project, the initial intuition was that the fact that the size of the window could only be reported in units of 2^n could lead to problems when a single octet is sent and the window that the receiving TCP entity reports must be adapted as a result of accepting this octet.

Our first thought was that there may be situations where a sending entity has a view of the size of the window at the receiving end that exceeds the maximum buffer space available. With the aid of our formal

specification, it is now easy to see why this is not the case. As highlighted in chapter 2, both entities maintain the send and receive window as 32-bit numbers. Besides this 32-bit number, they maintain a scale factor by which they right/left-shift the value reported in/taken from an acknowledgement segment. Note that this shift by a factor k, has the same effect as a floored division or multiplication by a factor 2^k.

Assume a receive buffer of capacity 2^{n+1}. By the fact that the receive buffer has capacity 2^{n+1}, we have a window size of at most 2^n and may use a scale factor $k \leq (n-1)$, resulting in a division or multiplication by 2^k. Now, if the receiver receives a segment carrying $0 < m \leq 2^n$ bytes, two scenarios may occur:

- The reduced buffer space (receive window) is reported in the acknowledgement segment. In this case, the window size that is reflected is $\lfloor (2^n - m)/2^k \rfloor < 2^{n-k} < 2^n$
- The old buffer space (receive window) is reported. In this case, nothing changes and therefore the window size that is reflected in an acknowledgement segment is $\lfloor 2^n/2^k \rfloor \leq 2^{n-k} < 2^n$

From this, it follows that the reported buffer size is always $\leq 2^{n-k}$ and therefore can never become greater than $2^{(n-k)+k} = 2^n$ when it is left-shifted at the remote end. Hence, a sending entity never has a view of the receive buffer space available at a receiving entity that exceeds the maximum buffer space available.[12]

The second problem of which we thought that it could arise relates to the fact that [36] explicitly states that a TCP window should not 'shrink' its receive window, meaning that the buffer capacity is reduced and therefore, the right edge of the window is moved to the left. An example of such shrinking is that a receiving entity first reports a receive window of size 4, and after receiving one octet reports a receive window of size 2. In this case, the window is shrunk by one octet.

The problem with shrinking the window is as follows: assume a sender and receiver that have agreed upon window of size 4. The sender first sends two octets and immediately thereafter, it sends another segment carrying the next two octets. By the time the first segment arrives at the receiver, it puts the octets in its receive buffer and, unfortunately, at the same time the capacity of the buffer is also reduced by one octet, causing the receiving entity to report a window of size 1 back to the sender instead of the expected window of size 2.

The problem arises because the second segment, carrying two octets, is already in transit. As soon as it arrives at the receiver, it will be discarded because of the fact that it contains more octets than the receiver may possibly accept. The sender will keep retransmitting this segment and it will be discarded as long as no octets are taken from the receive buffer, causing delay in the communication. If window sizes, and therefore the size of the octets that are sent, get bigger, such delay may have a significant impact on the performance of the protocol. Eventually, however, the octet will be accepted when buffer space becomes available as octets that arrived earlier are taken from the receive buffer.

When window scaling is in effect, one might expect such a scenario to occur every time an odd number of bytes is sent, as a result of the fact that the size of the window can only be reported in multiples of 2^n. There is an important distinction, however: in this case the actual capacity of the receive buffer is not reduced and the receiver maintains the window size as a 32-bit rather than a 16-bit number. Therefore, the second segment that may have been in transit already, will still be accepted and an acknowledgement containing the latest size of the window will be sent back within reasonable time. In a situation where the segment was not yet sent, the difference in the number of octets that may be sent is only 1, causing a performance rather than a correctness issue. We believe that a difference of one octet in the size of the send window does not have a significant impact on the performance of the protocol in the grander scheme of things.

[12] Note that it may still have a view that is too optimistic, more specifically in the second scenario where the old window size is reported in the acknowledgement segment.

Ambiguities in RFC 793

During the course of our project, there were several ambiguities in [36] that posed a problem to our specification efforts. As we expect that these ambiguities may also challenge the specification efforts of others, we will shortly summarise their nature and the solutions that we chose to solve them.

The first ambiguity concerns the window management suggestions as given on page 43. First of all, it is suggested that the sender could *"avoid sending small segments by waiting until the window is large enough before sending data"*. Great care should be taken if this feature is implemented, as it may easily lead to a deadlock, especially if data transfer is unidirectional. A scenario where the sending entity has sent $n - 1$ octets worth of its window which are all acknowledged but not yet forwarded to the application layer is easily constructed. Now, the sender would wait sending its last octet as a result of this optimisation, while a reopening of the window by the receiver is never communicated back to the sender since no acknowledgements are sent.

Additionally, the specification is very unclear about what window information to include in an acknowledgement segment: the actual size of the window – representing the available capacity of the receive buffer – or a larger window. In the latter case, we believe that the assumption is that it is highly likely that additional octets are consumed from the receive buffer during the round-trip time, effectively increasing the size of the receive window. As discussed in chapter 4, we strictly interpreted the requirements and always include the actual size of the window in our acknowledgements, but allow for a delay such that (newly arrived) octets may be consumed from the receive buffer before an acknowledgement is sent. Nothing is said in the specification about the amount of delay that is reasonable for an acknowledgement, the only requirement is that it is not *"undue"*.

Another ambiguity concerns the `RCV_WND` variable in the TCB. The RFC suggests that the value of this variable is adjusted as the size of the window changes, while at the same time requiring that the total of `RCV_WND` and `RCV_NXT` is not reduced. It is unclear whether the total may not be reduced when an incoming segment is processed, or not at all. Either way, we believe that it relates to the requirement that the right edge of the window should never be moved to the left. Therefore, as discussed in chapter 4, we chose to introduce an additional variable `RCV_RD_NXT` and keep `RCV_WND` constant at all times. This ensures that we keep track of the original size of the receive window throughout the execution of the protocol.

Regarding how to deal with zero windows, the RFC states on page 42 that: *"The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. [...] This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other."* The latter part of this statement is confusing, as the TCP instance will always retransmit segments that are on the retransmission queue even if the send window is zero. It is the first transmission of a segment even when the send window is zero that is crucial here. If this segment would not be transmitted, there would be nothing to retransmit and a deadlock may be the result where a sending TCP is never informed of a reopened window at the receiving side.

In general, it took us quite some time to distil the precise sliding window implementation from the specification. We believe that a formal specification as given in this thesis would have been more clear in the first place.

5.3 Formal verification of connection teardown

5.3.1 A model of TCP with connection teardown

In the previous section, we already discussed that we had to consider connection teardown separately because of the fact that it had a significant impact on the size of the state space. However, a verification of connection teardown only, without any involvement of data transfer, did not seem satisfactory to us, especially because of the fact that an important aspect of connection teardown is the fact that connections are closed in a simplex fashion, as discussed in chapter 2. Therefore, we opted for a verification of

Connection Teardown where one of the two TCP entities is required to send one octet of data before it could close its connection. Combined with our earlier verification, that showed that all octets that are buffered at the sender are eventually received, we believe that this scenario suffices to show that a connection will not be closed before all data is delivered.

Again, we had to prevent deadlock scenarios and undesired behaviour as a result of the reuse of sequence numbers from occurring in the state space as generated from our model. To understand that this problem arises again, recall from our discussion that the `FIN` segments also consume sequence numbers. The solution is now also easy to understand: rather than using $n+1$ sequence numbers to send n octets, we uses $n+3$ sequence numbers to account for the sequence numbers used for the `FIN` segments.

We obtained a model for TCP with connection teardown from our *SystemSpecification* in a similar fashion as in the previous section, although now we add $\{$`AL1_call_CLOSE`$|$`TCP1_rcv_CLOSE` = `TCP1_CLOSE`, `AL2_call_CLOSE`$|$`TCP2_rcv_CLOSE` = `TCP2_CLOSE`$\}$ to Θ to ensure that these actions only occur in a synchronised fashion.

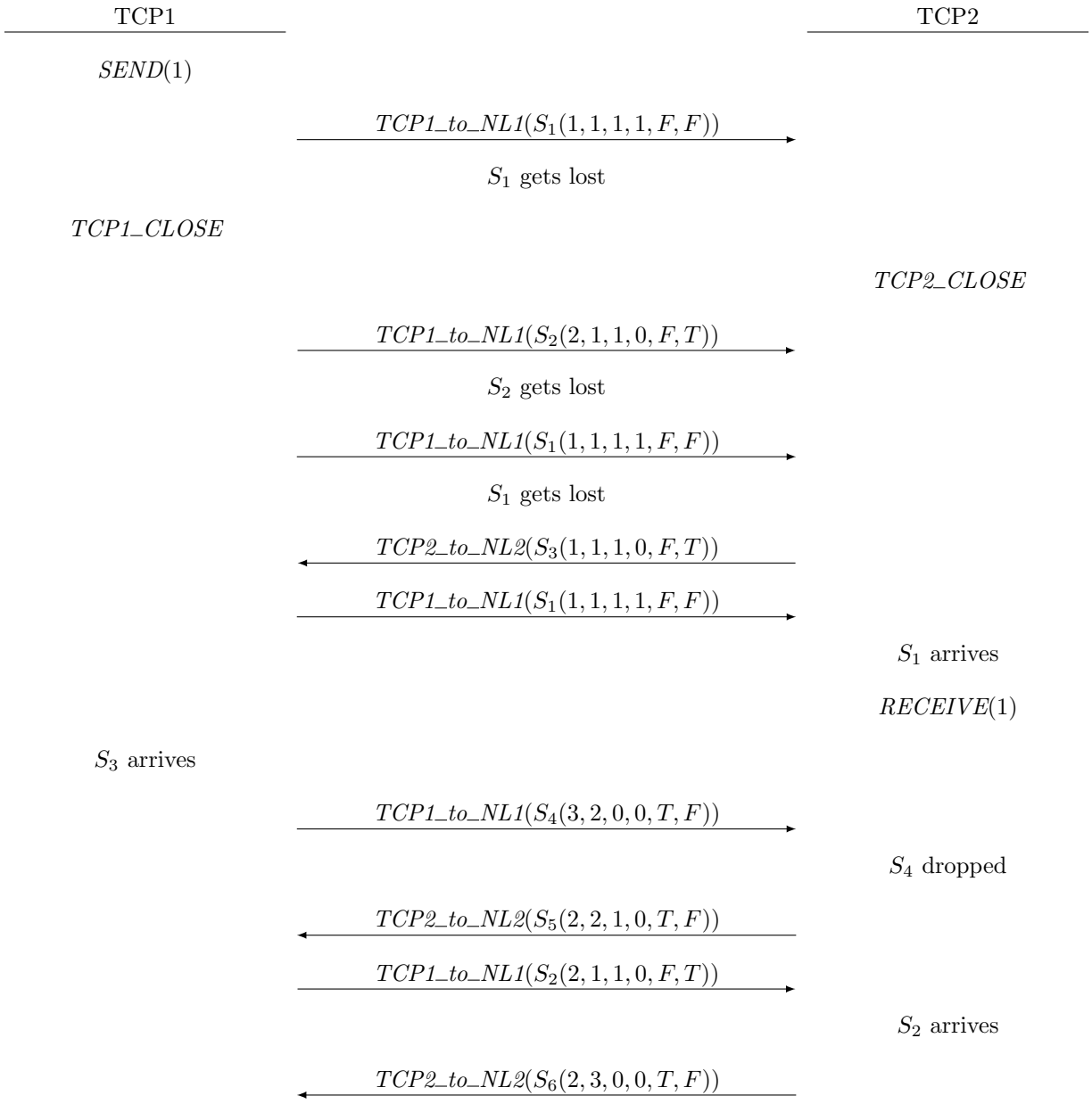


Figure 5.5: Livelock scenario in the connection termination procedure

However, analysis showed that our connection teardown procedure contained a trace as shown in figure 5.5 that leads to a livelock. As a result of the arrival of S_6, TCP1 progressed to the `TIME_WAIT` state and subsequently – as a result of the fact that we abstract away from the time wait timeout – to the `CLOSED` state. However, TCP2 was still waiting for an acknowledgement of its `FIN` segment S_3. The acknowledgement was sent (S_4) but dropped by TCP2 because its sequence number was not the sequence number that was expected. As a result of this, TCP2 was indefinitely stuck in the `CLOSING` state, retransmitting its `FIN` segment over and over again, for which an acknowledgement never arrived as TCP1 had already deleted all information related to the connection.

Clearly, this is a flaw in our modelling, as on page 22 of [36] it is stated that the `TIME_WAIT` state *"represents waiting for enough time to pass to be sure the remote TCP received the acknowledgement of its connection termination request"*. In other words, the connection must be kept 'half-open' as long as the remote entity may try to retransmit its `FIN` segment, to ensure that for each of these `FIN` segments that is received an acknowledgement is subsequently sent back and that therefore, eventually, the acknowledgement will arrive.

To fix this flaw in our model, we included the actions `TCP1_TW_TIMEOUT` and `TCP2_TW_TIMEOUT` of summand 4.13 in both δ and Θ, to ensure that they may only occur in a synchronised fashion. In other words: we required both TCP1 and TCP2 to arrive in the `TIME_WAIT` state before the connection could progress to the `CLOSED` state. However, as can be seen from figure 2.3, this solved only half the problem, as not all connections progress through the `TIME_WAIT` state during the closing procedure: they might as well progress through the `LAST_ACK` state. Therefore, we introduced an additional state `LAST_ACK2` and adapted summand 4.9 such that the TCP instance will progress from `LAST_ACK` to `LAST_ACK2` rather than `CLOSED`. Finally, we adapted summand 4.13 as follows:

$$+ \quad tcp\_TW\_TIMEOUT \cdot TCP(CLOSED, t) \ \lhd \ s = TIME\_WAIT \vee s = LAST\_ACK2 \ \rhd \ \delta \tag{5.2}$$

This modification ensures that during the closing procedure, each TCP entity will have to reach either the `TIME-WAIT` or `LAST_ACK2` state before the connection can be definitely closed. It turned out that this fixed our modelling flaw without the need to add timing aspects to our model.

5.3.2 Verification & results

First of all, it is important to state that by the fact that our model does not include connection establishment, the connection teardown procedure is only verified starting from the `ESTABLISHED` state. Scenarios where a connection is closed during connection establishment, before both ends have reached the `ESTABLISHED` state are not included.

As discussed in chapter 2, connections are closed in a simplex fashion. If one of the entities closes its connection, indicating that it has no more data to send, it must still accept segments from the remote end, and not progress to the `CLOSED` state until the other end has also indicated it has no more data to send.

Recall that (i) the TCP instance only accepts a `CLOSE` call from the application layer if it has no more octets to send, (ii) the `FIN` segment has a sequence number $\geq i + 1$ if i was the sequence number of the last data octet that was sent over the connection, and (iii) a receiving TCP instance only accepts segments that have a sequence number $i =$ `RCV.NXT`. Since the `CLOSED` state can only be reached after accepting a `FIN` segment (see figure 2.3), we can conclude from these facts that a TCP instance will never reach the `CLOSED` state without having accepted all data octets that were buffered in the send buffer at the other end at the time that the `CLOSE` call was issued.

However, this does not yet guarantee that connections will be closed whenever an application layer issues the `CLOSE` call to its TCP instance. To verify whether this is the case, we had to check several properties on the state space generated from our model of TCP with connection teardown.

The first of these properties, formulated as a regular μ-calculus formula, states that whenever TCP1

accepts the CLOSE call from the application layer, our model will eventually end up in a state from which it may perform the CONNECTION_CLOSED transition. From our discussion of our specification, we know that this transition is only enabled if both TCP instances are in the CLOSED state. As a reference, see summand 4.14 and recall that the TCP1_idle and TCP2_idle synchronise to the CONNECTION_CLOSED action as shown in figure 5.1.

$$[\mathtt{T}^* \cdot \mathtt{TCP1\_CLOSE}]\mu X \cdot (\langle \mathtt{T} \rangle \mathtt{T} \wedge [\neg \mathtt{CONNECTION\_CLOSED}]X) \tag{5.3}$$

We verified the same for TCP2. Taken together, these properties intuitively state *"whenever either of the TCP instances accepts the* CLOSE *call from the application layer, the connection will eventually end up in the* CLOSED *state"*. Hence, it is a *liveness* property.

In addition, we verified that both entities must accept the CLOSE call from their application layer before the connection may be closed. To this end, we checked the following *safety* property:

$$[(\neg \mathtt{TCP1\_CLOSE})^* \cdot \mathtt{CONNECTION\_CLOSED}]\mathtt{F} \tag{5.4}$$

stating that the connection can never reach a state in which it can perform a CONNECTION_CLOSED transition if TCP1 does not accept the CLOSE call from its application layer. Again, we also verified this for TCP2. Taken together, these properties ensure that both entities must accept the CLOSE call from their application layer before the connection ends up in the CLOSED state. Finally, we verified that our state space does not contain deadlocks as these again signal a problem for the same reasons as discussed before.

We used the `lpo2lps-dist` tool of the `ltsmin` toolset to generate a state space from the linear process equation obtained from our μCRL specification, in a distributed fashion. State space generation was again performed on 8 DAS-4 nodes equipped with an Intel Sandy Bridge E5-2620 processor clocked at 2.0 GHz, 64 gigabytes of memory and a K20m Kepler GPU with 6 gigabytes of on-board memory. It took around two minutes to generate a state space consisting of 42 levels, 3.296.792 states and 11.010.169 transitions. During state space generation, it was verified that there are no deadlocks.

After the state space generation, we again minimised the state space modulo branching bisimilarity, using the `ltsmin-reduce-dist` tool, and finally, we checked the aforementioned properties using the CADP toolset. All four properties, as well as absence of deadlocks, could be proved for our model.

Summary

Several issues surfaced when we started generating state-spaces for different parameterisations of our model. In this chapter, we first discussed these issues and explained that they forced us to split our verification into a separate verification of TCP's data transfer phase and connection teardown procedure.

For each of these verifications, we had to make some additional adaptations to our *SystemSpecification* to overcome issues with state space generation or unwanted behaviour due to modelling issues. We discussed these modifications in detail and showed how to obtain a model for unidirectional data transfer and connection teardown.

We showed that the external behaviour of the data transfer phase of unidirectional TCP is branching bisimilar to a FIFO queue and explained why this is sufficient to ensure the correctness of TCP extended with the Window Scale Option. We altered our specification to include undesired behaviour and showed that our verification approach succeeds at identifying these errors. With the aid of our formal specification, we were able to construct an extensive argument on the correctness of the window scale option. Finally, we highlighted some ambiguities in RFC 793 and gave some suggestions to alleviate them.

We also showed that if two TCP entities are in the ESTABLISHED state and either of them accepts the CLOSE call from the application layer, the connection will eventually be closed. Additionally, we showed that the connection can only be closed if both entites accept the CLOSE call from the application layer.

Chapter 6

Conclusions and future work

TCP plays an important role in the internet, providing reliable transport of data over possibly faulty networks. The protocol is complex and its specification consists of many documents that mainly describe the proposed functioning of the protocol in natural language. We set out to formally specify TCP extended with the Window Scale Option and verify its correctness, triggered by a concern that Dr. Barry M. Cook, CTO of 4 Links Limited, relayed in an e-mail about the correctness of this extension to TCP.

In a literature study that we executed as a preparation for our study, we found that some efforts had already been devoted to a formal specification and verification of both TCP and the Sliding Window Protocol. However, none of these efforts included the Window Scale Option. Furthermore, many efforts did not conform to the context in which TCP is generally implemented.

We took an incremental approach to our specification, starting with a very basic protocol and extending it with additional features during the course of this project. Because of the complexity of the protocol, we had to make a distinction between features that were essential for the correctness of the protocol and features that were proposed to improve the performance of the protocol. Features in the first category were all included in our specification. During the course of creating our specification, we found several ambiguities in RFC 793 that we have listed alongside suggestions to alleviate them. Because of its formal nature, we believe that our specification may serve as a useful reference for implementors of the protocol.

While it was sometimes difficult to interpret the specifications, the process algebra that we used for our specification, μCRL, turned out to be powerful enough to mimic the required features, albeit with slight adaptations. The size of the state space, however, turned out to be a limiting factor that forced us to split our verification efforts into a verification of TCP data transfer and a separate verification of connection teardown.

During our specification efforts, we discovered that there is no protection against sequence number reuse other than an assumption on the maximum lifetime of segments, combined with a transmission rate that ensures that all sequence numbers occupying the 'first half' of the window drain during the time that the 'second half' of the window is being transmitted. This discovery surprised us, as we initially thought that the Sliding Window Protocol provided a protection against the errors related to sequence number reuse as long as the size of the window never exceeded n in the presence of $2n$ sequence numbers. It turns out that the Sliding Window Protocol only protects against these errors if the communication mediums do not allow reordering.

With the aid of our μCRL specification and the `ltsmin` toolset, we were able to formally verify that our specification of unidirectional TCP extended with the Window Scale Option does not contain deadlocks, and that its external behaviour is branching bisimilar to a FIFO queue for a significantly large instance. Furthermore, we showed that the connection teardown procedure is successfully completed once both sides have indicated that they want to close the connection. From these facts, we conclude that the addition of the Window Scale Option does not negatively affect the correctness of the protocol. We have only verified this correctness for a relatively small instance of the protocol but believe that the

specification is general enough to make the introduction of errors as parameters are increased highly unlikely.

While our verification only entails unidirectional data transfer, our specification also supports bidirectional data transfer. Therefore, we think that the main contribution of this work is a formal, unambiguous specification of the main features of TCP. It may be interesting to repeat our verification efforts in a few years time, as the available computational power has increased, to see if state space generation for bidirectional TCP has become feasible by then.

Our specification could serve as a basis for a more complete specification of TCP. In chapter 4, we listed several features defined in RFC 793 and RFC 1122 that are not yet included. Of these features, especially piggy-backed acknowledgements, which requires a modification in the way that incoming segments are processed, is of interest as this feature involves the data transfer phase of TCP. Furthermore, including performance enhancements such as the optimisations as mentioned in RFC 1122, are also of interest as implementing them would bring our specification closer to real-world implementations.

While connection establishment could also be added to the specification, we think that this is of less importance given the great deal of attention that this aspect already received in the literature. Efforts directed at a verification of TCP's connection management should rather focus on the connection teardown procedure, since enabling this procedure in the specification at all times has significant effects on the size of the state space. We expect that further development of formal verification tools and pushing them to the limit could bring within reach a verification that more closely mimics a real-world situation.

Finally, we found that we had to use several different toolsets that all have shortcomings in some respect. The tool that we used to generate state spaces in a distributed fashion, `ltsmin`, lacked support in the generation of counter-examples or traces to deadlock situations, while `mCRL2` included this support but had shortcomings in dealing with specification, state space generation and property checking. As a result of this, we ended up specifying in μCRL and converting between several formats to work with our specification in both `ltsmin` and `mCRL2`, which made the process rather cumbersome from time to time.

Appendix A

Workflow

Local

1. Specify model using μCRL
2. Linearise model using μCRL
   `mcrl -tbfile -regular Model`
3. Convert μCRL model to LPS
   `tbf2lps Model.tbf Model.lps`

DAS-4

1. Generate LTS using `lpo2lts-dist` or `lps2lts-dist`
   `prun -t 20:0 -v -np 8 -native '-l fat,gpu=K20' -1 -sge-script $LTSMIN/etc/prun-openmpi-ltsmin $LTSMIN/bin/lpo2lts-dist --deadlock --cached --stats --when --where Model.tbf Model.dir`
2. Minimise LTS modulo branching bisimilarity using `ltsmin-reduce-dist`
   `prun -t 20:0 -v -np 2 -4 -sge-script $LTSMIN/etc/prun-openmpi-ltsmin $LTSMIN/bin/ltsmin-reduce-dist -b --stats --when Model.dir Model-min.dir`
3. Compare the LTSs using `ltsmin-compare` to an LTS of our behavioural specification to see whether they are branching bisimilar
   `ltsmin-compare -b Model-min.dir BehaviouralSpec.dir`
4. Or: First convert LTSs to format that the `mCRL2` toolset understands (.aut)
   `ltsmin-convert --rdwr --segments 1 Model-min.dir Model-min.aut`
5. Copy .aut files to local system

Local

1. Compare .aut files of TCP and Behavioural specification using the `mCRL2` toolset, to see whether they are branching bisimilar
   `ltscompare Model-min.aut BehaviouralSpec.aut --equivalence=branching-bisim --preorder=unknown --verbose`

Appendix B

Axioms of process algebra

The axioms presented in this appendix are kindly provided by Wan Fokkink.

A1	$x + y$	$=$	$y + x$	
A2	$x + (y + z)$	$=$	$(x + y) + z$	
A3	$x + x$	$=$	x	
A4	$(x + y)\cdot z$	$=$	$x\cdot z + y\cdot z$	
A5	$(x\cdot y)\cdot z$	$=$	$x\cdot(y\cdot z)$	
A6	$x + \delta$	$=$	x	
A7	$\delta\cdot x$	$=$	δ	
CM1	$x \parallel y$	$=$	$(x \lfloor\!\lfloor y + y \lfloor\!\lfloor x) + x \mid y$	
CM2	$a(\vec{d}) \lfloor\!\lfloor x$	$=$	$a(\vec{d})\cdot x$	
CM3	$a(\vec{d})\cdot x \lfloor\!\lfloor y$	$=$	$a(\vec{d})\cdot(x \parallel y)$	
CM4	$(x + y) \lfloor\!\lfloor z$	$=$	$x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	
CM5	$a(\vec{d})\cdot x \mid b(\vec{e})$	$=$	$(a(\vec{d}) \mid b(\vec{e}))\cdot x$	
CM6	$a(\vec{d}) \mid b(\vec{e})\cdot x$	$=$	$(a(\vec{d}) \mid b(\vec{e}))\cdot x$	
CM7	$a(\vec{d})\cdot x \mid b(\vec{e})\cdot y$	$=$	$(a(\vec{d}) \mid b(\vec{e}))\cdot(x \parallel y)$	
CM8	$(x + y) \mid z$	$=$	$x \mid z + y \mid z$	
CM9	$x \mid (y + z)$	$=$	$x \mid y + x \mid z$	
CF	$a(\vec{d}) \mid b(\vec{d})$	$=$	$c(\vec{d})$	if $a \mid b = c$
CF′	$a(\vec{d}) \mid b(\vec{e})$	$=$	δ	if $\vec{d} \neq \vec{e}$ or a and b do not communicate
C1	$x \lhd \mathtt{T} \rhd y$	$=$	x	
C2	$x \lhd \mathtt{F} \rhd y$	$=$	y	

DD	$\partial_H(\delta)$	$=$	δ	
D1	$\partial_H(a(\vec{d}))$	$=$	$a(\vec{d})$	if $a \notin H$
D2	$\partial_H(a(\vec{d}))$	$=$	δ	if $a \in H$
D3	$\partial_H(x+y)$	$=$	$\partial_H(x)+\partial_H(y)$	
D4	$\partial_H(x{\cdot}y)$	$=$	$\partial_H(x){\cdot}\partial_H(y)$	
CD1	$\delta \between x$	$=$	δ	
CD2	$\delta \mid x$	$=$	δ	
CD3	$x \mid \delta$	$=$	δ	
SUM1	$\sum_{d:D} x$	$=$	x	
SUM2	$\sum_{d:D} P(d)$	$=$	$\sum_{d:D} P(d) + P(d_0) \quad (d_0 \in D)$	
SUM3	$\sum_{d:D}(P(d)+Q(d))$	$=$	$\sum_{d:D} P(d) + \sum_{d:D} Q(d)$	
SUM4	$(\sum_{d:D} P(d)){\cdot}x$	$=$	$\sum_{d:D}(P(d){\cdot}x)$	
SUM5	$(\sum_{d:D} P(d)) \between x$	$=$	$\sum_{d:D}(P(d) \between x)$	
SUM6	$(\sum_{d:D} P(d)) \mid x$	$=$	$\sum_{d:D}(P(d) \mid x)$	
SUM6′	$x \mid (\sum_{d:D} P(d))$	$=$	$\sum_{d:D}(x \mid P(d))$	
SUM7	$\partial_H(\sum_{d:D} P(d))$	$=$	$\sum_{d:D} \partial_H(P(d))$	
SUM8	$(\forall\, e{\in}D\; P(e) = Q(e))$	$\Rightarrow$	$\sum_{d:D} P(d) = \sum_{d:D} Q(d)$	
SUM9	$\rho_f(\sum_{d:D} P(d))$	$=$	$\sum_{d:D} \rho_f(P(d))$	
SUM10	$\tau_I(\sum_{d:D} P(d))$	$=$	$\sum_{d:D} \tau_I(P(d))$	
R1	$\rho_f(\delta)$	$=$	δ	
R2	$\rho_f(\tau)$	$=$	τ	
R3	$\rho_f(a(\vec{d}))$	$=$	$f(a)(\vec{d})$	
R4	$\rho_f(x+y)$	$=$	$\rho_f(x)+\rho_f(y)$	
R5	$\rho_f(x{\cdot}y)$	$=$	$\rho_f(x){\cdot}\rho_f(y)$	
B1	$x{\cdot}\tau$	$=$	x	
B2	$x{\cdot}(\tau{\cdot}(y+z)+y)$	$=$	$x{\cdot}(y+z)$	
TID	$\tau_I(\delta)$	$=$	δ	
TI1	$\tau_I(a(\vec{d}))$	$=$	$a(\vec{d})$	if $a \notin I$
TI2	$\tau_I(a(\vec{d}))$	$=$	τ	if $a \in I$
TI3	$\tau_I(x+y)$	$=$	$\tau_I(x)+\tau_I(y)$	
TI4	$\tau_I(x{\cdot}y)$	$=$	$\tau_I(x){\cdot}\tau_I(y)$	

Bibliography

[1] B. Badban, W.J. Fokkink, and J. van de Pol. Mechanical verification of a two-way sliding window protocol. In P.H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *CPA*, volume 66 of *Concurrent Systems Engineering Series*, pages 179–202. IOS Press, 2008.

[2] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μcrl. *The Computer Journal*, 37(4):289–307, 1994.

[3] J. Billington and B. Han. On defining the service provided by tcp. In M.J. Oudshoorn, editor, *ACSC*, volume 16 of *CRPIT*, pages 129–138. Australian Computer Society, 2003.

[4] J. Billington and B. Han. Closed form expressions for the state space of tcp's data transfer service operating over unbounded channels. In V. Estivill-Castro, editor, *ACSC*, volume 26 of *CRPIT*, pages 31–39. Australian Computer Society, 2004.

[5] J. Billington and B. Han. Formalising tcp's data transfer service language: a symbolic automaton and its properties. *Fundamenta Informaticae*, 80(1):49–74, 2007.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In R. Guérin, R. Govindan, and G. Minshall, editors, *SIGCOMM*, pages 265–276. ACM, 2005.

[7] S. Blom and S. Orzan. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[8] R. Braden. Requirements for internet hosts-communication layers. *RFC 1122*, 1989.

[9] D. Chkliaev, J. Hooman, and E.P. de Vink. Verification and improvement of the sliding window protocol. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2003.

[10] E. Emerson and J. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[11] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[12] J.C.A. De Figueiredo and L.M. Kristensen. Using coloured petri nets to investigate behavioural and performance issues of tcp protocols. 1999.

[13] S. Floyd, J. Mahdavi, M. Mathis, and A. Romanow. Tcp selective acknowledgment options. *RFC 2018*, 1996.

[14] W.J. Fokkink. *Introduction to process algebra*. Texts in theoretical computer science. Springer, 2000.

[15] W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, and J. van de Pol. Verifying a sliding window protocol in îîîîîî. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2004.

[16] International Organization for Standardization/International Electrotechnical Commission et al. Information technology—open systems interconnection—basic reference model: The basic model. *ISO/IEC*, pages 7498–1, 1994.

[17] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer, 1990.

[18] B. Han. Formal specification of the tcp service and verification of tcp connection management. *PhD Thesis*, 2004.

[19] B. Han and J. Billington. Validating tcp connection management. *Proceedings of the conference on Application and theory of petri nets: formal methods in software engineering and defence systems-Volume 12*, pages 47–55, 2002.

[20] B. Han and J. Billington. Formalising the tcp symmetrical connection management service. pages 178–184, 2003.

[21] B. Han and J. Billington. Experience using coloured petri nets to model tcp's connection management procedures. pages 57–76, 2004.

[22] B. Han and J. Billington. Termination properties of tcp's connection management procedures. In G. Ciardo and P. Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2005.

[23] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. *RFC 1323*, 1992.

[24] K. Jensen. Coloured petri nets. *Petri nets: central models and their properties*, pages 248–299, 1987.

[25] D. Kozen. Results on the propositional μ-calculus. In M. Nielsen and E.M. Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1982.

[26] M. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.

[27] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in lotos. In K.R. Parker and G.A. Rose, editors, *FORTE*, volume C-2 of *IFIP Transactions*, pages 495–510. North-Holland, 1991.

[28] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[29] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[30] S.L. Murphy and A.U. Shankar. Service specification and protocol construction for the transport layer. In *SIGCOMM*, pages 88–97, 1988.

[31] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[32] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known tcp implementation problems. *RFC 2525*, 1999.

[33] D. Peled. *Software Reliability Methods*. Springer, 2001.

[34] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[35] J. Postel. Internet protocol. 1981.

[36] J. Postel. Transmission control protocol. *RFC 793*, 1981.

[37] T. Ridge, M. Norrish, and P. Sewell. A rigorous approach to networking: Tcp, from implementation to protocol to service. In J. Cuéllar, T.S.E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[38] I. Schieferdecker. Abruptly-terminated connections in tcp–a verification example. *Proceedings of the COST*, 247:136–145, 1996.

[39] M.A. Smith and K.K. Ramakrishnan. Formal specification and verification of safety and performance of tcp selective acknowledgement. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.

[40] M.A.S. Smith. Formal verification of communication protocols. In R. Gotzhein and J. Bredereke, editors, *FORTE*, volume 69 of *IFIP Conference Proceedings*, pages 129–144. Chapman & Hall, 1996.

[41] M.A.S. Smith. Formal verification of tcp and t/tcp. *PhD Thesis*, 1997.

[42] W. Stevens, M. Allman, and S. Paxson. Tcp congestion control. *RFC 2581*, 1999.

[43] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[44] A.S. Tanenbaum. *Computer networks (4. ed.)*. Prentice Hall, 2002.

[45] J.L.A. van de Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17, 1995.

[46] R.J. van Glabbeek. The linear time - branching time spectrum ii. In E. Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[47] R.J. van Glabbeek. The linear time-branching time spectrum i – the semantics of concrete, sequential processes. 2001.

[48] R.J. van Glabbeek and W. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.