Using Coloured Petri Nets to Investigate Behavioural and Performance Issues of TCP Protocols*

Jorge C. A. de Figueiredo[†‡] and Lars M. Kristensen[‡]

[†] Departamento de Sistemas e Computação, UFPB, Brazil
E-mail: abrantes@daimi.au.dk

[‡] CPN Group, Department of Computer Science, University of Aarhus, Denmark
E-mail: kris@daimi.au.dk

Abstract

This paper deals with modelling and analysis of the Transmission Control Protocol (TCP). The TCP protocol is the transport protocol used on the Internet. We present a generic Coloured Petri Net model (CPN model) suited for performance and behavioural analysis of different versions of the TCP protocol. The CPN model covers connection establishment and termination, data transfer phase, and concepts such as slow start, congestion avoidance, fast retransmit, and fast recovery.

As a representative example we show how the CPN model can be instantiated to analyse two different versions (TCP Reno and TCP Tahoe) of the TCP protocol. By means of simulations of the CPN model, we investigate the behaviour of both TCP versions when they face different packet loss situations, and we investigate the impact of packet loss on the throughput obtained with the TCP Reno and TCP Tahoe protocols.

1 Introduction

The Internet has over the past decade experienced a crescent and rapid growth. Despite the continuous and significant advances in Internet technology, severe congestion problems have appeared with the growth of Internet applications. One of the main causes for congestion problems is the transport protocol implementations [11]. The *Transmission Control Protocol (TCP)* is the transport protocol used on the Internet. Different versions of the TCP protocol have been defined [8, 10, 22] in order to cope with congestion problems.

The window flow control in TCP is one of the mechanisms for handling congestion on the Internet. Basically, the window flow control manages the demand on the receiver's capacity. Four intertwined algorithms have been defined and introduced in modern TCP versions. They can act as a preventive way to avoid congestion as well as a congestion recovery scheme to restore an operating state, when faced with unexpected changes such as an increase in traffic and loss of packets. These algorithms, named *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery*, have been successfully applied and widely adopted. Some of them are classified as mandatory for TCP implementations.

*This work was partially supported by CNPq/Brazil under grant 201462/91-5 (NV).

Many case studies have been conducted to investigate the performance of the different versions of TCP over congested networks. Kumar [17] uses a stochastic model to predict the throughput of different versions of TCP, considering the presence of random losses on a wireless link in a local area network. Lakshman and Madhow [18] examine the TCP performance over wide area networks when data traffic coexists with real-time traffic. Fall and Floyd [8] present the benefits of adding selective acknowledgements and a selective strategy to TCP. Fall and Floyd used simulation to make comparisons between different TCP versions. Goyal et al [9] study the design issues for improving TCP performance over an *ATM Unspecified Bit Rate (UBR)* service. Simulation is used to obtain different performance measures and TCP is analysed over different switch drop policies. Ost and Haverkort [21] use a Stochastic Petri Net model to evaluate performance of windowing mechanisms in world-wide web applications.

In this paper we apply hierarchical Coloured Petri Nets (CP-nets or CPNs) [12–14, 16] for modelling and simulation-based performance analysis of TCP protocols. For construction and simulation of the CPN models we use the Design/CPN tool [2, 20]. The Design/CPN tool has previously been used in a number of projects on performance analysis, e.g., in the areas of high-speed interconnects [3] and ATM networks [4, 5]. The reader is assumed to be familiar with the basic concepts of high-level Petri Nets [15].

The primary objective of this paper is to present a *generic* CPN TCP model to analyse the performance and the behaviour of TCP protocols. The model incorporates aspects such as *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery.* The model has been constructed in such a way that different TCP versions can be analysed. Performance analysis can be carried out by conducting lengthly simulations of the CPN model. Different kind of performance measures can be defined and observed. Such model and simulation-based performance analysis can be applied to investigate *what-if* questions, and used as a test-bed for evaluation of different aspects and variations of the TCP protocol. As a representative example, we compare the behaviour of the two most common TCP versions (*TCP Reno* and *TCP Tahoe*) when they face different packet loss situations. We also consider performance analysis for both TCP versions.

This paper is organised as follows. Section 2 introduces the basic concepts of TCP protocols with emphasis on the data transfer part and the concepts of *slow start*, *congestion avoidance*, *fast retransmit*, and *fast recovery.* Section 3 presents the developed CPN TCP model. Section 4 describes the simulation scenario. Section 5 presents the analysis performed for the *TCP Reno* and *TCP Tahoe* protocols for different packet loss situations. Finally, in Sect. 6 we sum up the conclusions.

2 The Transmission Control Protocol

The Transmission Control Protocol (TCP) provides a connection-oriented, reliable, byte stream service [6, 22]. A connection is initialised when two processes want to communicate. When a connection is established, the TCP protocol is able to transfer a continuous stream of octets. Octets are packed into segments[1] for transmission over the Internet. When the communication is completed, the connection is terminated. TCP connections are full duplex, i.e., messages can flow in both directions. The focus of this paper is on the data transfer part. Therefore, we do not consider the connection part in any detail. The reader interested in the connection part can refer to [7, 22].

[1] Throughout this paper we will use the terms *segment* and *packet* interchangeably.

During the data transfer part, each segment transfered in a TCP connection is given a unique sequence number. The sliding window strategy is employed to provide efficient transmission and flow control. The TCP sender keeps for each connection, the necessary information in order to guarantee the correct behaviour of the sliding window strategy. The TCP sender also maintains a variable that contains the maximum receiver window size indicating the buffering capacity of its counterpart.

The TCP receiver can accept out-of-order packets. The TCP receiver has a finite buffer where packets can be stored. The packets must be delivered in correct sequence to its TCP user. The TCP receiver returns an acknowledgement for every packet successfully received. The acknowledgement contains the sequence number for the next packet it expects and the current maximum window size the TCP receiver can cope. It is important to point out that the acknowledgements are cumulative, i.e., an acknowledgement with sequence number n indicates that all packets up to and including $n-1$ were successfully received.

TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system. Basically, when TCP sends a segment it maintains a timer waiting for an acknowledgement. The timer is set to a *Retransmission Time-Out (RTO)* value [6], which is defined based on a running estimate of the packet *Round-Trip Time (RTT)*. The segment is retransmitted if no acknowledgement is received within this time.

The basic ideas discussed above are common to all TCP versions. However, to improve TCP throughput, many modern TCP versions incorporate modified flow control procedures to limit the number of packets in the network. We discuss such procedures in the following subsections.

2.1 Slow Start and Congestion Avoidance

Slow start and *congestion avoidance* are two independent algorithms used to treat computer networks congestion problems [11]. Although, the two algorithms are independent and have different objectives, they are in practice implemented together [22, 23].

The basic idea behind *slow start* is that the packets rate injection into the network at the sender side is based on the rate at which acknowledgements are returned by the receiver side. Thus, instead of injecting multiple packets into the network as performed by old TCPs implementations, the sender starts by transmitting few segments[2]. As soon as it receives an acknowledgement, the number of segments to be sent is gradually increased. This prevents the sender from overwhelming the network with a large amount of traffic, which is likely to cause segment losses. Actually, two segments are allowed to be sent for each acknowledgement received.

A packet can be lost either by damage in transit or by congestion in the network. Losses due to damage are extremely rare [11]. So, it is assumed that a packet loss indicates congestion in the network. *Congestion avoidance* was defined as a way to cope with the loss of packets. Thus, if congestion is detected the sender must have some strategy to decrease its utilisation of network. Congestion is detected at the sender side by a timeout or by the reception of duplicate acknowledgements (*dupACKs*)[3].

[2] As originally defined, the sender starts transmission in the slow start phase by sending one segment [11]. However a larger initial window was already suggested and evaluated in [1].

[3] Notice that the concept of duplicate acknowledgment does not refer to the communication channel duplicating the acknowledgement.

To implement these algorithms, a congestion window size variable (*CWND*) is kept at the sender as a measure of the capacity of the network. Moreover, a slow start threshold variable (*SSTHRESH*) is maintained to distinguish between *slow start* and *congestion avoidance* phases. The maximum number of data that the sender can send is defined by the minimum of *CWND* and the receiver window size (*RemoteWindow*). If $CWND < SSTHRESH$, then the connection is in *slow start* phase. Otherwise, *congestion avoidance* is performed. The sender starts transmission in the *slow start* phase by sending one segment. When the sender receives an acknowledgement for a new segment, *CWND* is incremented by 1. This means that *CWND* is doubled every round trip time. Therefore, *slow start* phase corresponds to an exponential increase in the number of data which can be sent.

When congestion is detected, one-half of the current congestion window size is saved in $SSTHRESH$. The $SSTHRESH$ value should be at least two segments. Additionally, if congestion was triggered by a timeout, then *CWND* is set to 1 and the *slow start* phase starts.

During *congestion avoidance* phase, the sender increases its $CWND$ variable by $1/CWND$ every time an acknowledgement is received. Differently from *slow start*, a linear increase in the number of data allowed to be sent is observed in this phase.

2.2 Fast Retransmit and Fast Recovery

As stated previously, whenever the TCP receiver receives new data, it sends an acknowledgement to the TCP sender specifying the sequence number for the next expected packet. However, if an out-of-order packet is received (indicating a potential packet loss), a *dupACK* is immediately sent to the sender [23].

The idea behind *fast retransmit* is to realize, as soon as possible, if a segment has been lost. It is assumed that if the cause for the duplicate acknowledgements is just a reordering of segments, the number of duplicate acknowledgements is very small. Effectively, the sender waits for a fixed small number K of *dupACKs*. Typically, K is set to 3. If more than K *dupACKs* are received, the TCP sender concludes that the segment indicated in the duplicate acknowledgements has been lost. Immediately, the sender retransmits the segment what appears to be the segment lost. At this point, the sender reduces its $CWND$ variable by half plus K segments. The addition of K is based on the assumption that K more packets have successfully left the network. Also half of the original $CWND$ is saved to $SSTHRESH$. For each additional *dupACK*, the sender increases $CWND$ by one and sends a new segment whenever possible.

When a new acknowledgement is received for the retransmitted segment, *congestion avoidance* is performed instead of *slow start*. This enhancement is known as *fast recovery* and contributes to a higher throughput under moderate congestion. Thus, instead of setting $CWND$ to one segment as in a regular time-out situation, TCP sets $CWND$ to $SSTHRESH$ and performs *congestion avoidance*. This occurs approximately one *RTT* after the lost segment is retransmitted.

2.3 TCP Versions

Different TCP versions have been proposed. Basically, the various TCP versions differ in the way they recover from the loss of packets. Below we summarise a number of TCP versions. Details about the different versions can be found in [8–10, 17, 22].

TCP Vanilla: Incorporates *slow start* and *congestion avoidance* aspects. The recovery of a packet loss is performed in the original way, i.e., the TCP sender waits for a coarse timeout to retransmit the packet. After retransmission, *slow start* is performed.

TCP Tahoe: *Fast retransmit*, *slow start* and *congestion avoidance* are considered. This version tries to conclude as soon as possible that a segment has been lost. When more than K *dupACKs* are received, it behaves as if a timeout has occurred and begins retransmission. However, after retransmission, *slow start* is performed.

TCP Reno: Similar to *TCP Tahoe* but takes *fast recovery* into consideration. This means that instead of slow-starting, the TCP Reno sender makes some estimates of the amount of outstanding data upon reception of additional incoming duplicate acknowledgements. The *TCP Reno* version is said to be conservative [17] because it retransmits only one segment, even in case of multiple packet losses in one window.

TCP New Reno: A slight variation of *TCP Reno* that eliminates the approach in case of multiple losses. In such a situation, when the fast retransmission is first triggered, the sender saves the highest sequence number sent. When a new acknowledgement is received, it verifies if the acknowledgement includes all the segments sent. If so, the sender acts as in the *TCP Reno* version by setting $CWND$ to $SSTHRESH$ and performing *congestion avoidance*. On the other hand, if not all segments were acknowledged, the sender immediately retransmits the next segment that appears to be lost. This continues until all the segments are acknowledged.

TCP SACK: Incorporates the *selective acknowledgement* approach to efficiently recover from multiple segment losses [8]. In this version, the information carried in the acknowledgements is more elaborate and contains additional details about the segments which have been received by the TCP receiver. From this information, the TCP sender can infer about the segments that were not received by its counterpart.

3 The CPN TCP Model

This section contains a detailed description of the CPN TCP model. Considering that the primary purpose of TCP is to provide a reliable data transfer and connection oriented service between pairs of processes, four aspects were considered in the model as suggested in [7]: connection, basic data transfer, flow control, and reliability. The CPN TCP model is a timed hierarchical CPN model and was developed in such a way that different TCP versions can be analysed with only minor modifications to the CPN model.

Figure 1 provides an overview of the CPN TCP model. The CPN TCP model is divided in two main parts: the connection part (right part of the hierarchy page) and the data transfer

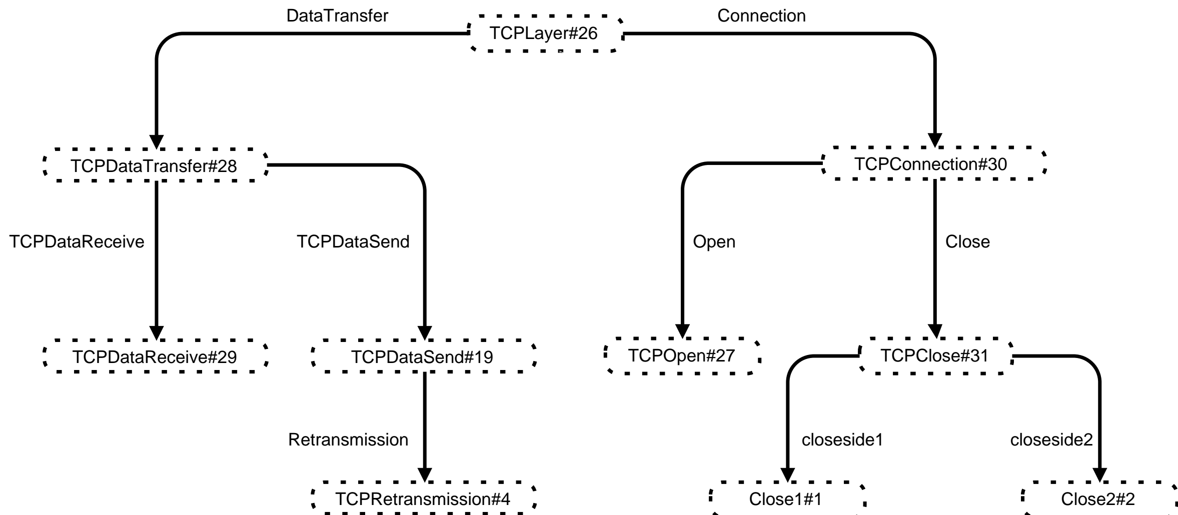

Figure 1: The Hierarchy page for CPN TCP model

part (left part of the hierarchy page). The TCP connection part is divided in two subparts. Page *TCPOpen* models the three-way handshake procedure used for TCP connection establishment. Page *TCPClose* and its subpages *Close1* and *Close2* model the termination of a TCP connection.

The data transfer phase is modelled by the subpages of *TCPDataTransfer*. During this phase, the two processes involved in the connection exchange messages. Recall that each message is packed in segment(s) which are transfered via the communication system. Segments are reassembled in the receiver side to reconstitute the original message. An acknowledgement is required for each segment successfully received. A TCP process is not only exclusively a sender or a receiver, i.e., each process must be able to act as a sender and also as a receiver. The TCP data transfer part is therefore divided in a data send part (pages *TCPData Send* and *TCPRetransmission*) and a data receive part (page *TCPDataReceive*).

Although a complete TCP model has been developed, we present a simplified version in this paper. Since the main objective of this paper is to analyse TCP performance and behaviour during the data transfer phase, the entire TCP connection phase was omitted. It is assumed that the connection is successfully initialised and all connection information accordingly set. The data transfer part of the model is described in the following subsections.

3.1 The Transmission Control Block

Much information is generated when a connection is opened. For example, the processes should agree on the initial sequence numbers. This information is updated when data transmission occurs. The *Transmission Control Block (TCB)* is the data structure where the connection information is stored [7]. In the CPN model, TCB is modelled by an accordingly named place. There is one token on this place for each of the currently open TCP connections. Figure 2 lists the definition of the colour set (type) *TCB*.

Id is a pair of endpoints that identifies a connection. *LocalWindow* determines the maximum number of bytes the process can receive. This information should be included in all data

```
color TCB = record Id: Connection *
                   LocalWindow: Int *
                   RemoteWindow: Int *
                   CWND: Int *
                   SSTHRESH: Int *
                   SN: Int *
                   SNm: Int *
                   RN: Int *
                   NACK: CongAvoid *
                   NSeg: Int *
                   RNStatus: Status *
                   ConState: States;
```

Figure 2: The TCB colour set

segments that are transmitted. *RemoteWindow* indicates the maximum number of bytes the remote process can receive. This information is used to determine the data segment size.

CWND and *SSTHRESH* keep current information about congestion window size and threshold, respectively. These values are used to perform *fast retransmit* and *fast recovery* as explained previously. *SN*, *SNm*, and *RN* are three attributes used to control the flow of segments and to define the sliding window. *SN* indicates the next segment to be sent. *SNm* indicates the sequence number of the segment its counterpart is waiting for. This information is important to determine if a new segment can be sent as well as in the retransmission mechanism. *RN* indicates the segment the local process is expecting to receive. The *NACK* attribute indicates if the connection is performing *fast retransmit*. Moreover, when receiving *dupACKs*, it keeps track of the number of *dupACKs* which have been received. *NSeg* is an auxiliary attribute used to update *CWND*, in case of *congestion avoidance*. *RNStatus* can assume two values: *acked* or *notacked*. As soon as a new segment is received, *RNStatus* is set to *notacked*. When the acknowledgement is sent, it changes to *acked*. An acknowledgement is sent only when *RNStatus* is set to *notacked*. *ConState* indicates either a connection phase or a data transfer phase.

3.2 Sending Segments

In the TCP model two different types of segments are defined: *ack segments* and *data segments*. *Ack segments* carry only the sequence number of the next expected segment. *Data segments* are more complex. Besides its sequence number, each *data segment* carries the remote window size information. Remember that since data can be sent in both directions, a data segment could be used to acknowledge receipt of a segment.

Figure 3 depicts page *TCPDataSend* modelling the transmission of segments. In this paper, the part of the model responsible for the fragmentation of data from the TCP user layer was simplified. It was assumed that the TCP sender is an infinite TCP source, i.e., TCP always sends a segment whenever allowed by the window. All segments are also assumed to have the same size. The part of page *TCPDataSend* modelling access to *TCB* has been highlighted using dashed lines and arcs.

Occurence of transition *Accept* (top) indicates the acceptance of a new segment to be sent,

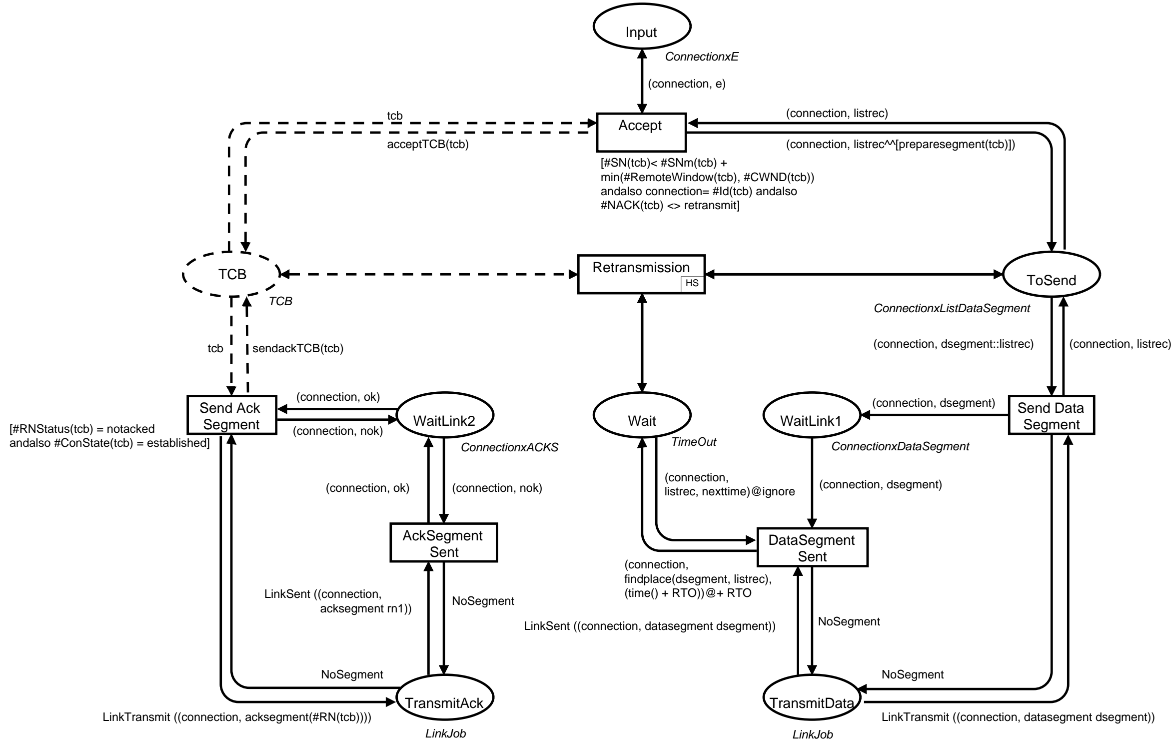

Figure 3: The *TCPDataSend* page

i.e., the window is not full. According to the sliding window strategy, a new segment will be accepted if its sequence number is less than the last sequence number acknowledged plus the minimum value of the remote window and congestion window sizes. The guard of transition *Accept* guarantees that this condition is satisfied. If transition *Accept* occurs, a new (token) segment is created by the function *preparesegment* and deposited on place *ToSend*. Also, TCB is updated by function *acceptTCB*. Place *ToSend* keeps a list of data segments to be sent. A FIFO policy is adopted.

Occurence of transition *Send Data Segment* (right) represents that the segment is being enqueued in the transmission link buffer. A copy of the data segment is kept on place *WaitLink1*. When the *data segment* is effectively enqueued, transition *DataSegment Sent* occurs and a token is deposited on place *Wait*. Place *Wait* maintains a list of *data segments* sent that will be used for retransmission when necessary (subpage *Retransmission*). This is a timed place and the Retransmission Time Out (*RTO*) value is used to determine the timestamp for its tokens.

Occurence of transition *Send Ack Segment* (left) represents the sending of an *ack segment*. This is possible when the status for the reception of segments is *notacked*, as indicated in the guard of the transition. After sending an *ack segment*, the status is set to *acked* by the function *sendackTCB*. Place *WaitLink2* and transition *AckSegment Sent* has a similar functionality as *WaitLink1* and *DataSegment Sent*.

Figure 4 depicts page *Retransmission*. This it the subpage of the substitution transition *Retransmission* in Fig. 3. Two different forms of retransmission are modelled: a timeout retransmission or a fast retransmission. Transition *Timeout* models the regular retransmission,

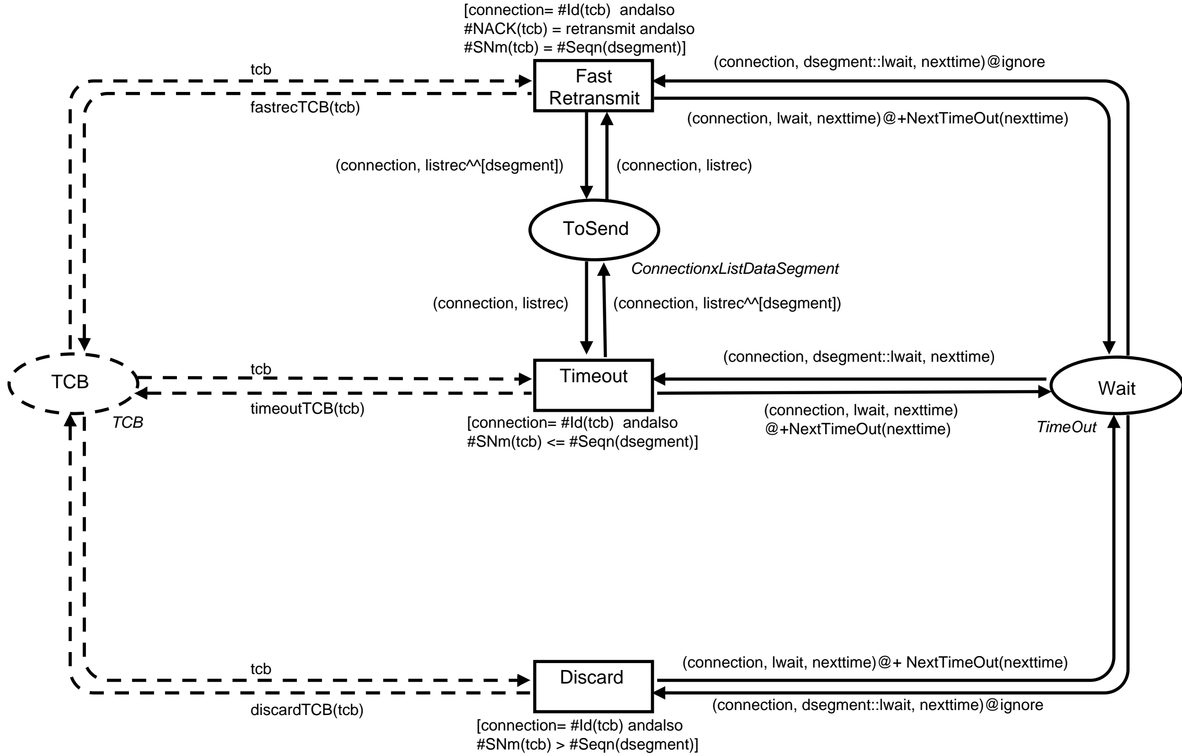

Figure 4: The *Retransmission* page

i.e., when no acknowledgement is received and the timeout timer expires. After occurence of transition *Timeout*, the TCB attributes are updated accordingly by the function *timeoutTCB*. For example, *CWND* and *SSTHRESH* are set to 1 and half of the current *CWND*, respectively.

The TCP *fast retransmit* aspect is modelled by the transition *Fast Retransmit*. This transition occurs only when K *dupACKs* have been received. When the *NACK* attribute in *TCB* has the value *retransmit*, indicating that K duplicate acknowledgements have been received, transition *Fast Retransmit* can occur. Function *fastrecTCB* updates *CWND* and *SSTHRESH* as described in Sect. 2.2. Transition *Discard* removes segments from the list on place *Wait* whenever segments are acknowledged.

3.3 Receiving Segments

Page *TCPDataReceive* modelling the reception of segments is shown in Fig. 5. Transitions *Receive Ack Segment* (right) and *Receive Data Segment* (left) model the reception of *ack segments* and *data segments*, respectively. When an *ack segment* is received, function *ackrecTCB* updates *TCB*. Minor adjustments in function *ackrecTCB* are required in order to consider different versions of TCP.

If a *data segment* is received the TCB information is updated by the function *datarecTCB*. Moreover the segment received is stored on the place *Segments Received*. A special function *updatelr* is used in order to verify if the segment was already received or if it was a new one. This function also reassembles the messages before they are delivered to upper protocol

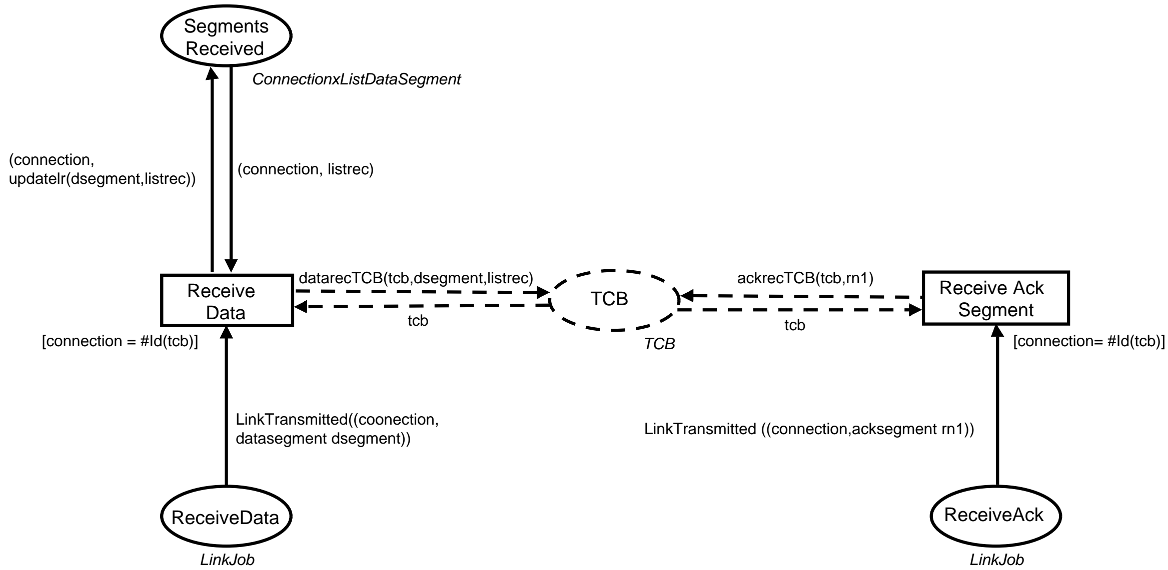

Figure 5: The *TCPDataReceive* page

layers.

4 Simulation Scenario and Model

The simulation scenario considered was based on the scenario used in [8] by Fall and Floyd to compare different TCP versions.

Figure 6 shows the network topology for the considered simulation scenario, i.e., the environment in which we study the TCP Reno and the TCP Tahoe protocols. The bandwidth between the *Sender* and the *Switch* is $8Mbps$ (Mbit per second) and the link delay is $0.1ms$. The bandwidth and delay for the link between the *Switch* and the *Receiver* are $0.8Mbps$ and $100ms$, respectively.

Data are sent exclusively in one direction, from the *Sender* to the *Receiver*. A finite-buffer drop tail switch was considered. Also, three TCP connections from the *Sender* to the *Receiver* was considered, however only the first connection was analysed. The other two connections were used only to achieve the desired pattern of drops to be analysed (1, 2 or 3 packet losses in one window of packets). The pattern of packet drops is changed by the number of packets sent by the second and third connections. Thus, simulations and analysis were performed considering three situations: one packet loss (packet number 14), two packet losses (packets 14 and 28) and three packet losses (packets 14, 26 and 28).

Figure 7 provides an overview of the model considered for simulation. It represents an in-

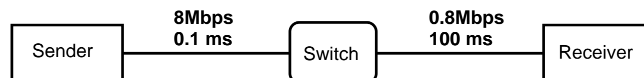

Figure 6: Simulation scenario

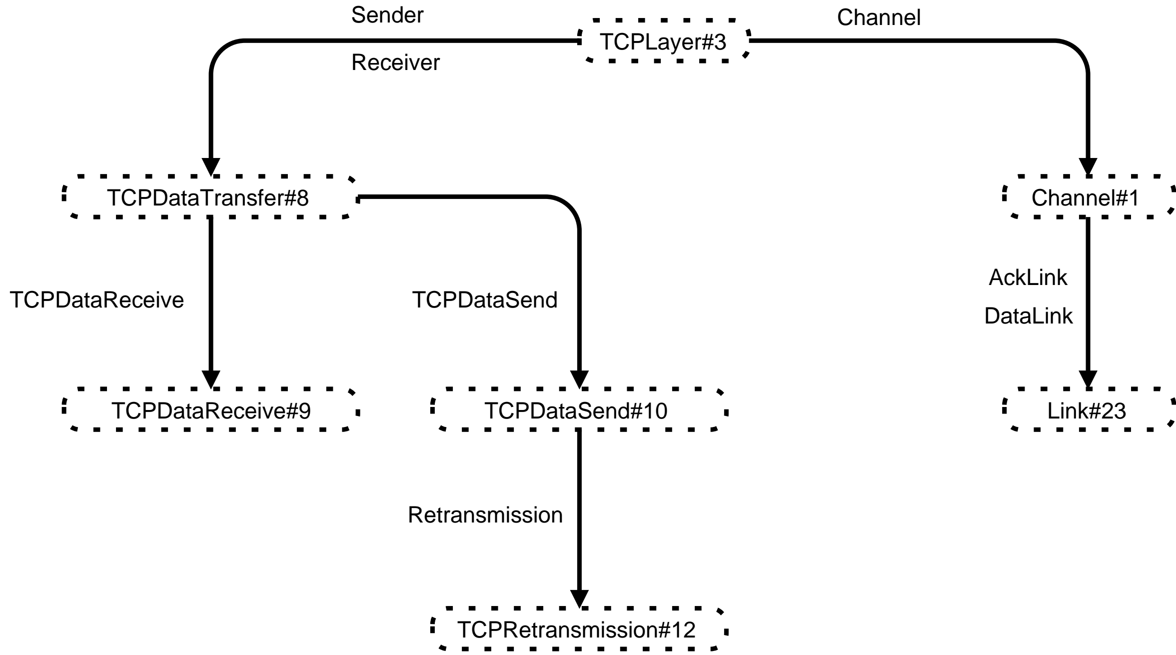

Figure 7: The hierarchy page for the simulation model

stantiation of the CPN TCP model presented in the previous section in such a way that the two TCP versions can be studied. The model consists of two instances of page *TCPDataTransfer*: one representing the *Sender* and one representing the *Receiver*. In page *TCPDataReceive* the function *ackrecTCB* was defined differently for the two TCP versions. In case of TCP Reno, *ackrecTCB* was set to define *fast retransmit* and *fast recovery* concepts. For the TCP Tahoe, *ackrecTCB* does not consider *fast recovery*. In both cases, the congestion window threshold K was set to 3.

The switch and link bandwidth capacities and delays are modelled by the pages *Channel* and *Link*. Figure 8 depicts page *Link*. Packets to be transmitted are represented by *Link-Transmit* tokens in place *Incoming Packets*. Occurence of transition *Packet Arrival* indicates that the packet was transmitted. Function *LinkResourceJobArrival* checks the availability of space in the buffer. If there is no space left, the packet is dropped. A *LinkSent* token is deposited back in place *Incoming Packets* with timestamp set to the transmission time value. This indicates that it is necessary to wait at least the transmission time to be able to transmit a new packet. Transition *Schedule Packet* occurs whenever the timestamp of the token in place *LinkResource* is satisfied. This timestamp is set based on the link delay value.

The Design/CPN tool [20] was used to simulate the model. The performance facilities [19] of the Design/CPN simulator were used to define the analysis functions and to perform data collection. Basically two functions were defined: a function that keeps track of all data sent by the TCP sender and a function that determines the number of packets successfully delivered at the receiver side.

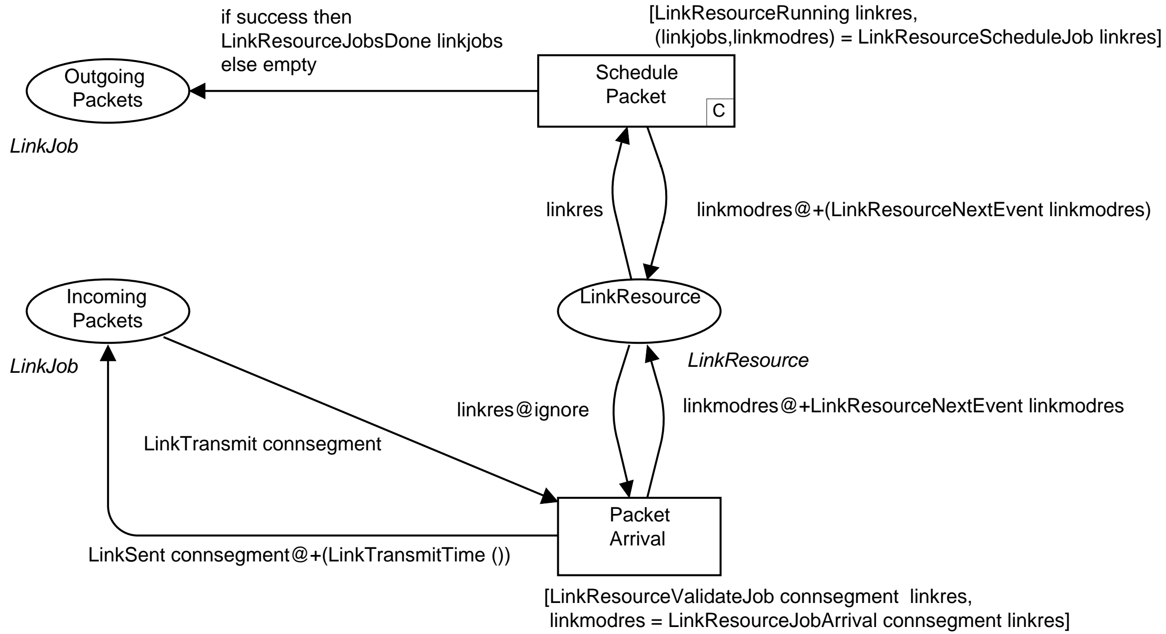

Figure 8: CPN model for the link

5 Analysis of the Model

Simulation was performed considering one, two and three packet loss situations. For each situation, the sequence of packets sent by the TCP sender was observed. In addition, the number of packets delivered was obtained and a performance measure was computed.

5.1 Behavioural Analysis

To observe the behaviour of the two TCP versions, we defined a graph that shows the sequence of packets sent from the *Sender* to the *Receiver*. We have time on the x-axis, and the packet number (mod 60) on the y-axis. A square represents each packet as it arrives in the switch. Packet losses are represented in the figure by a cross mark.

One Packet Loss

We consider the situation where the packet with sequence number 14 is lost. Figures 9 and 10 show the sequence of packets sent by the *Sender*, considering the *TCP Reno* and *TCP Tahoe* versions respectively.

In case of *TCP Reno*, packets 0-13 are sent without problems (14 square marks in the four initial windows in Figure 9). During this phase, TCP performs slow start and the congestion window increases exponentially from 1 to 15. Packet number 14 is dropped, represented by a cross mark in Fig. 9. This means that all the packets for the fourth window were successfully delivered, except packet 14. Thus, the TCP sender receives 7 *ack segments* and it increases *CWND* from 8 to 15. The TCP sender is allowed to send packets 15-28 (fifth window in Figure 9).

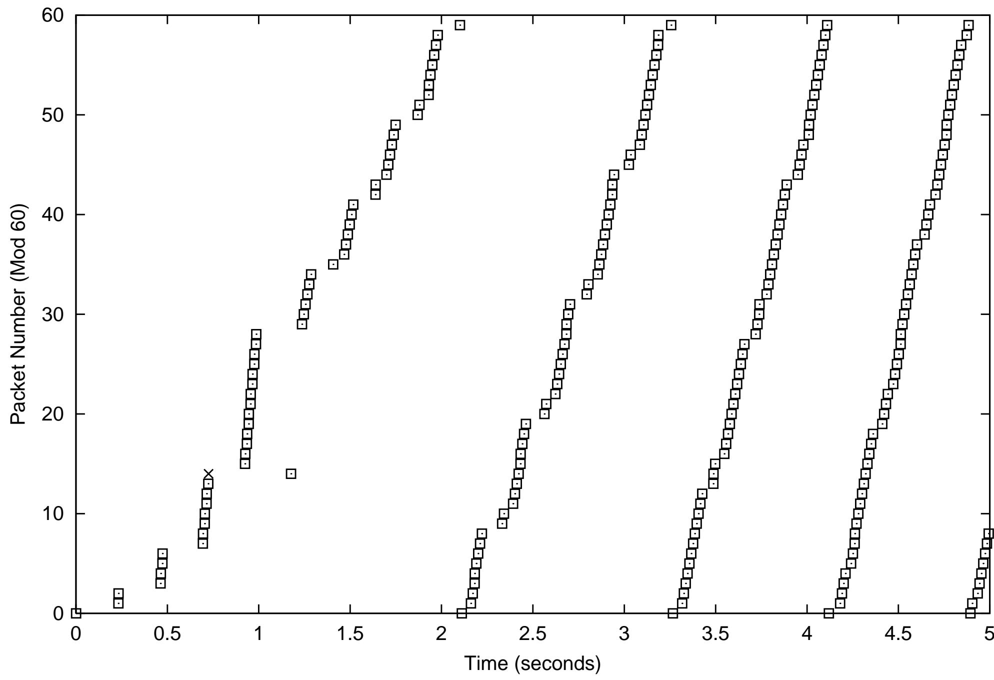

Figure 9: TCP Reno - one packet loss

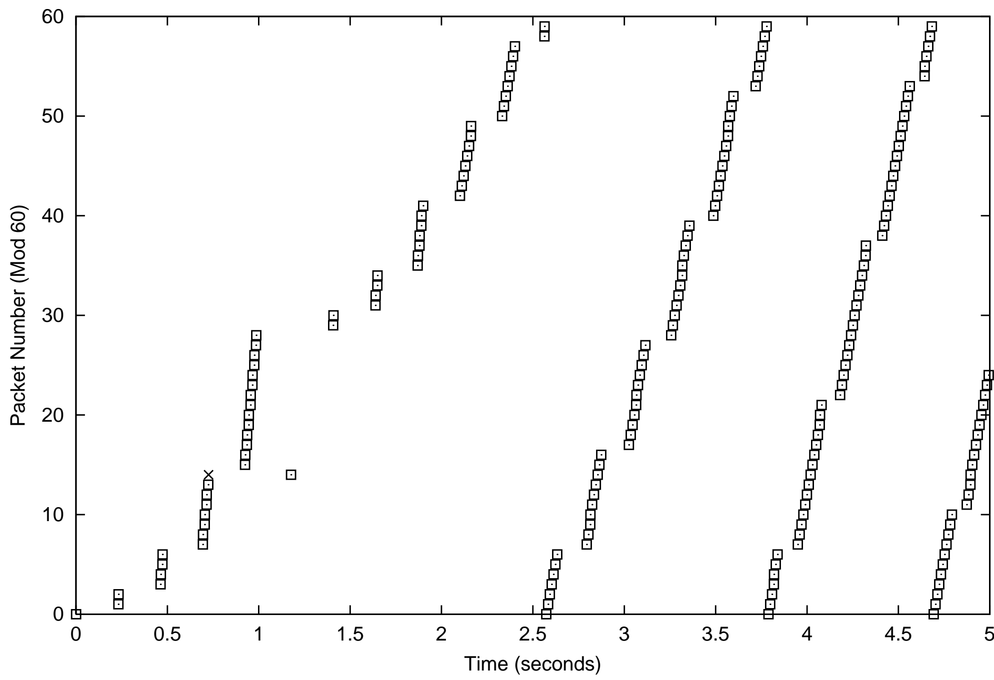

Figure 10: TCP Tahoe - one packet loss

When the receiver successfully receives packet number 13, it acknowledges reception by asking for packet number 14. Packets 15-28 were successfully delivered. Therefore, besides the first acknowledgement asking for packet 14, the sender receives 14 additional acknowledgements for this same packet (*dupACKs*).

After reception of the third *dupACK* asking for packet 14, fast retransmission is performed. At this point, *SSTHRESH* is set to 7 (half of *CWND*) and *CWND* is set to 10 (half plus K). Packet 14 is then retransmitted. Since 11 *dupACKs* were received after retransmission of packet 14, *CWND* is increased up to 21.

When *CWND* is increased for the last six *dupACKs*, the sender is allowed to send packets 29-34. When the retransmitted packet is received, the sender exits fast recovery and starts congestion avoidance with *CWND* set to 7.

In case of *TCP Tahoe*, the behaviour is identical to *TCP Reno's* behaviour until the reception of the third *dupACK* for packet 14. Then, the sending *TCP Tahoe* reduces it congestion window to one and retransmits packet 14. When an acknowledgement is received for the retransmitted packet, *CWND* is increased by 1 and packets 29 and 30 are sent. The sender continues in *slow start* and when packet 34 is sent, the slow-start threshold is reached and *congestion avoidance* starts.

Two Packet Losses

Figures 11 and 12 show the sequence of packets sent for the two packet loss situation considering *TCP Reno* and *TCP Tahoe*, respectively. The two lost packets are 14 and 28, as shown by the cross marks in Figures 11 and 12.

The protocol behaves exactly the same as in the one drop situation until packet 28 is sent. Since packet 28 is also lost, the number of duplicate acknowledgements asking for packet 14 is 13 instead of 14 for the one-drop situation.

Considering the *TCP Reno* protocol, the sender is able to send packets 29-33. After reception of the third duplicate acknowledgement, *CWND* drops to 10 and increases up to 20 due to the reception of the last 7 duplicate acknowledgements asking for packet 14. When the retransmitted packet 14 is acknowledged, packet 28 is expected. Since this is the first acknowledgement that asks for packet 28, the sender is allowed to send a new packet (packet 34). At this point, the sender exits fast recovery with *CWND* of 7.

Since packet 28 is also lost, the sender will receive 6 duplicate acknowledgements asking for packet 28. Reception of the third duplicate acknowledgement triggers a second fast retransmission situation. *CWND* and *SSTHRESH* are set to 6 and 3 respectively. *CWND* increases to 9 when the sender receives the sixth duplicate acknowledgement. Then, the sender is able to send packets 35 and 36. When the acknowledgement for the second retransmitted packet is received, the sender exits again fast recovery with a congestion window of three.

In the case of *TCP Tahoe* protocol, after the reception of the third *dupACK* for packet 14, *CWND* is reduced to 1 and packet 14 is retransmitted. The sender then receives an acknowledgement asking for packet 28 and *CWND* is increased by one. Packets 28 and 29 are sent and the sender continues in the *slow start* phase. The *congestion avoidance* phase starts when the sender sends packet 40.

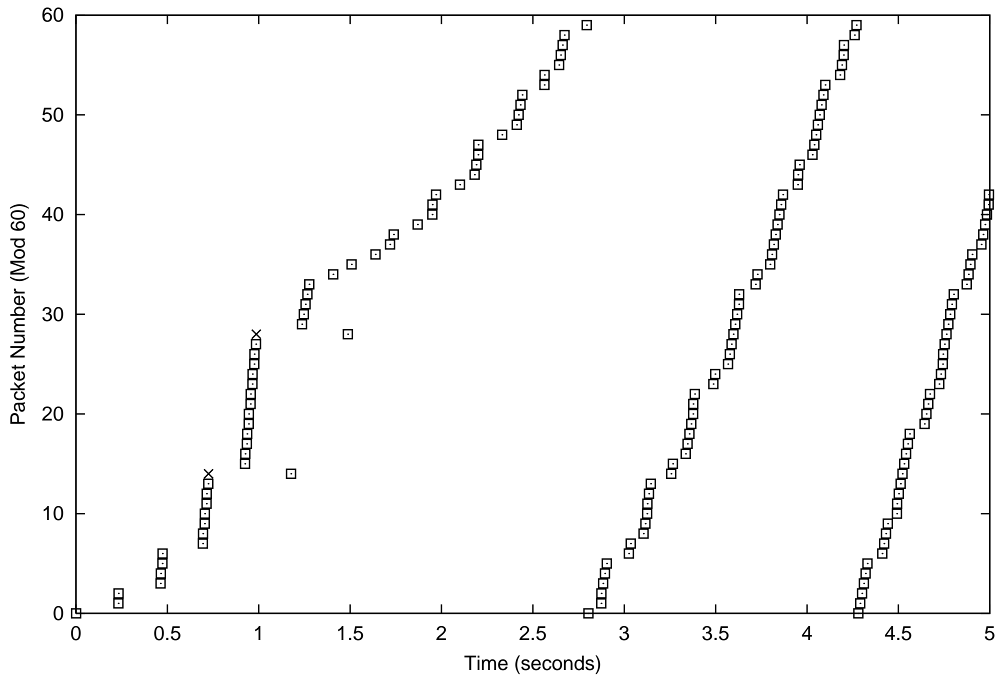

Figure 11: TCP Reno - two packet losses

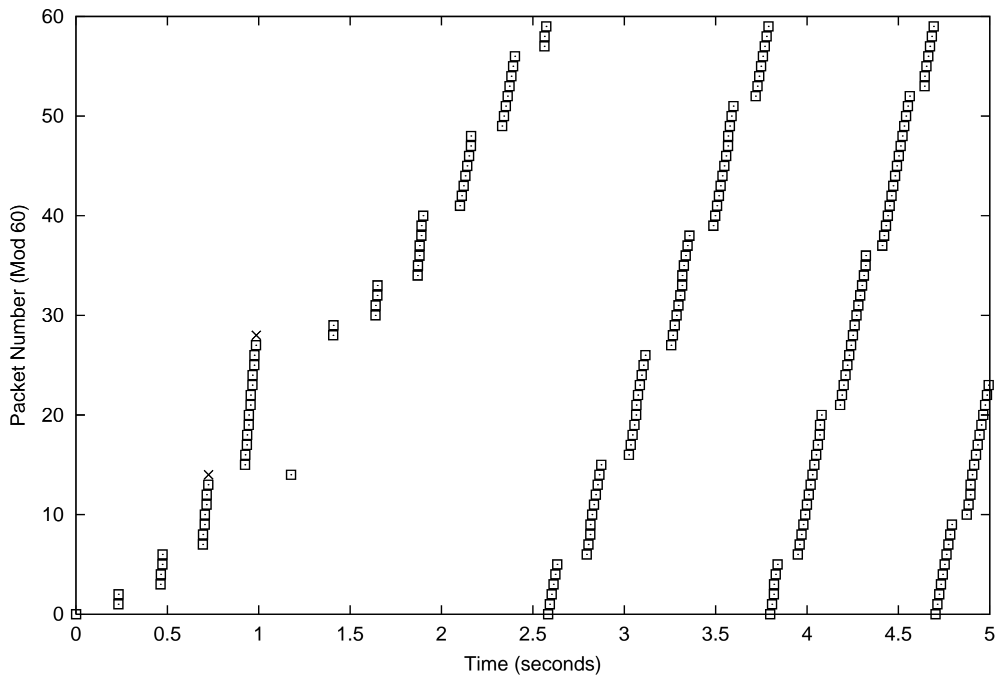

Figure 12: TCP Tahoe - two packet losses

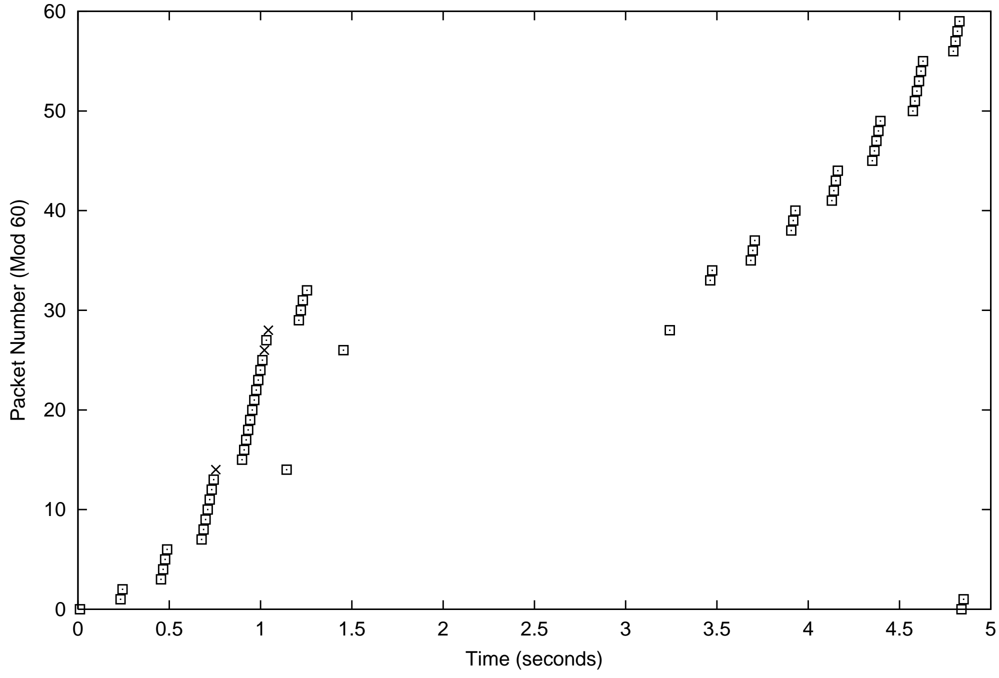

Figure 13: TCP Reno - three packet losses

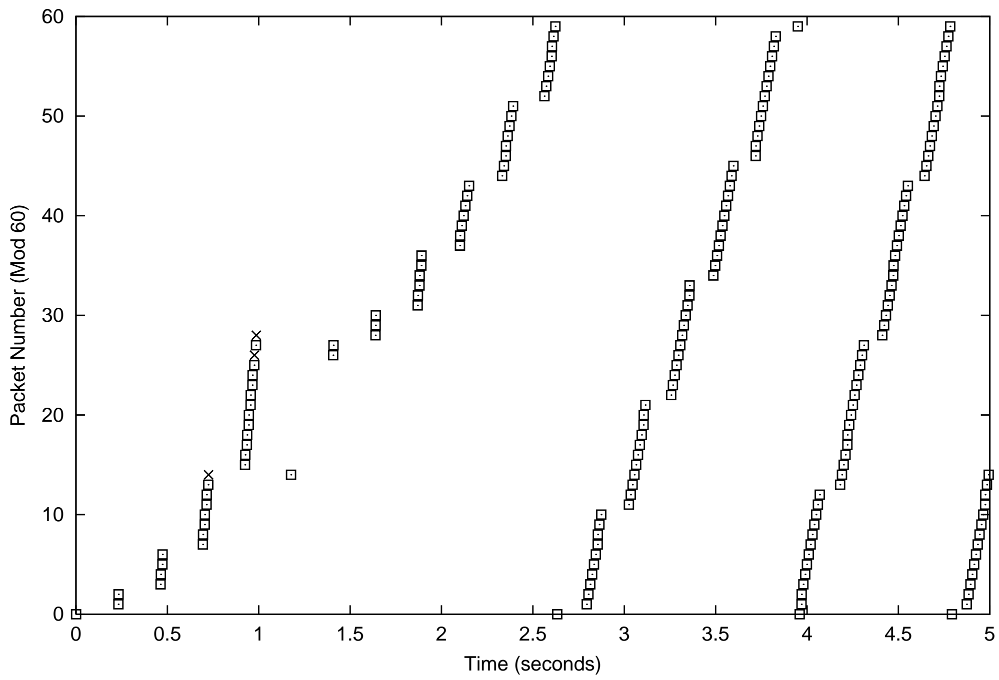

Figure 14: TCP Tahoe - three packet losses

Three Packet Losses

In addition to loss of packets 14 and 28, now also packet 26 is lost. Figure 13 shows the sequence of packets sent by the *TCP Reno* sender for the three packet loss situation. Figure 14 shows the sequence of packets for the *TCP Tahoe* protocol.

In case of *TCP Reno*, packets 0-13 are sent without problems. When packets 15-28 are sent, the TCP sender receives 12 *dupACKs* asking for packet 14. After retransmission of packet 14, *CWND* decreases to 10 and reaches 19 due to *dupACKs*. When the TCP sender receives an acknowledgement asking for packet 26, it exits *fast recovery*. At this point, *CWND* is set to 7. However, since packet 26 is lost, after receiving three *dupACKs* asking for packet 26, the TCP sender performs a second fast retransmission. *SSTHRESH* is reduced to 3 and *CWND* is set to 6. The three next *dupACKs* increase *CWND* to 9. After receiving an acknowledgement asking for packet 28, the sender exits *fast recovery* and *CWND* is reduced to 3.

Since *CWND* is 3 and three packets still unacknowledged, the sender is stalled and is unable to perform fast retransmission. This is reflected in Fig. 13 as the absence of squares in the period from 1.5 seconds to about 3 seconds. Thus, the sender waits for a retransmission timeout. When timeout occurs, packet 28 is retransmitted and the TCP sender sets *CWND* to 1, and *slow start* is then performed. The congestion window *CWND* now increases exponentially to a value of 3, after which congestion avoidance is entered since *SSTHRESH* is 3.

TCP Tahoe acts as in the two previous cases to cope with the loss of packet 14. When the sender receives an acknowledgement asking for packet 26, the *CWND* is increased to 2. In some situations, *TCP Tahoe* sender may forget the fact that some packets were previously sent. This problem is likely to appear when multiple packet losses occur in one window of packets. So, packets 26 and 27 are sent for the second time. Two acknowledgements for packet 28 are received. However the congestion window is increased to 3 because only one of the two acknowledgements represents the acknowledgement for a new data. The sender remains in the *slow start* phase and switches to *congestion avoidance* phase when packet 37 is sent.

5.2 Performance Analysis

Different kind of performance measures can be defined and analysed by simulation of the CPN TCP model. In this section we consider efficiency, which is defined as the number of packets delivered to the receiver divided by the maximum number of packets that could be delivered (when there is no loss of packets). It means that efficiency is 1 when there is no packet drop.

In the simulations we have assumed a segment size of 1000 bytes and a TCP maximum receiver window size of 32K bytes. All simulations runs correspond to 5 seconds of real time.

We observed the efficiency in both TCP versions for the situations of packet losses presented in the previous section. Table 1 summarises the results.

The results indicate that the *TCP Tahoe* keeps almost the same efficiency for the three situations. If we analyse the *TCP Tahoe* behaviour presented in the last section, we can observe that *TCP Tahoe* recovers from the packet losses in the three situations performing one *slow start*.

TCP Reno presents greater efficiency when it faces the one packet loss situation. This

	Efficiency	
Situation	TCP Reno	TCP Tahoe
1 packet loss	0.59	0.47
2 packet losses	0.38	0.47
3 packet losses	0.14	0.45

Table 1: Efficiency Results

occurs because Reno uses *fast recovery* after retransmission of the loss packet, allowing a smoothly recover from the packet loss. For the two packet losses situation, *TCP Reno* was forced to perform two successive *fast retransmit* and *fast recovery* procedures. Thus, the congestion window was reduced in half twice. Since the reductions were performed in two successive round-trip times, the performance of TCP connection was degraded. *TCP Reno* presents vary poor performance (efficiency of 14%) for the three packet losses situation. In this case, *TCP Reno* waited for a retransmit timeout to recover from a packet loss and a considerably performance reduction was observed.

The experiments showed that *TCP Reno* performs nicely in the case of a simple packet drop. For two or more packet losses the performance of *TCP Reno* is significantly degraded.

6 Conclusion

We have presented a timed hierarchical CPN model for behavioural and performance analysis of TCP protocols. The model considers the four most prevalent TCP algorithms for congestion control. We have used the term *generic* to emphasise that with minor changes it is possible to adapt the model for different TCP versions.

Simulation based performance analysis for the *TCP Reno* and *TCP Tahoe* protocols was also presented focusing on efficiency under different packet loss scenarios. The obtained results indicate that the model can be successfully applied to investigate TCP performance and can be used as a test-bed to evaluate different scenarios and variations of TCP.

The CPN TCP model has also been used as a building block in two other performance analysis projects. In one project it was used as a component in a model focusing on capacity planning and performance analysis of web servers. The TCP protocol is used in this context for transmission of documents between web clients and web servers. In another project, it was used as component in a model analysing the performance of a web based application for distributed teaching. The TCP protocol is used in this context to implement the remote procedure call mechanism by means of which clients and servers communicate.

Although the preliminary results indicate that the CPN TCP model can be applied to analyse performance of TCP protocols, further investigation on this issue still needed. The idea is to use the newly developed batch simulation facilities in the Design/CPN tool in order to make a larger number of simulations, considering various network scenarios and different switch drop policies.

Further refinements in the CPN TCP model to incorporate additional aspects are also important. For example, a more detailed model for *RTO* estimation can be developed. The timer granularity is an important factor in determining TCP performance.

References

[1] M. Allman, C. Hayes, and S. Ostermann. An Evaluation of TCP Slow Start Modifications. *Computer Communication Review*, 28(3):41–52, 1998.

[2] S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 1997.

[3] G. Ciardo, L. Cherkasova, V. Kotov, and T. Rokicki. Modeling a Scaleable High-Speed Interconnect with Stochastic Petri Nets. In *Proceeding of PNPM'95*, pages 83–93. IEEE Computer Society Press, 1995.

[4] H. Clausen and P. R. Jensen. Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets. In *Proceeding of PNPM'93*, pages 280–289. IEEE Computer Society Press, 1993.

[5] H. Clausen and P. R. Jensen. Analysis of Usage Parameter Control Algorithms for ATM Networks. In S. Tohmé and A. Casada, editors, *Broadband Communications II (C-24)*, pages 297–310. Elsevier Science Publishers, 1994.

[6] D. E. Comer. *Internetworking with TCP/IP, Vol. 1, Principles, Protocols and Architecture.* Prentice Hall, 1994.

[7] DARPA. Transmission Control Protocol — Protocol Specification. RFC — 793, September 1981.

[8] K. Fall and S. Floyd. Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3):5–21, 1996.

[9] R. Goyal, R. Jain, S. Kalyanaraman, S. Fahmy, and B. Vandalore. Improving the Performance of TCP over the ATM-UBR Service. *Computer Communications*, 21(10):898–911, July 1998.

[10] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of SIGCOMM'96*. ACM, August 1996.

[11] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM'88*, pages 314–329. ACM, August 1988.

[12] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[13] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1995.

[14] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[15] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets*. Springer-Verlag, 1991.

[16] L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):98–132, 1999.

[17] A. Kumar. Comparative Performance Analysis of Versions of TCP in a Local Network with a Lossy Link. *IEEE/ACM Transactions on Networking*, 6(4):485–498, August 1998.

[18] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwith-Delay Products and Random Loss. *IEEE/ACM Transactions on Networking*, 5(3), June 1997.

[19] B. Lindstrøm and L. M. Wells. Simulation Based Performance Analysis in Design/CPN. In K. Jensen, editor, *Proceedings of Workshop on Practical Use of Coloured Petri nets and Design/CPN*, pages 117–130, 1998.

[20] Design/CPN Online. http://www.daimi.au.dk/designCPN/.

[21] A. Ost and B. R. Haverkort. Analysis of Windowing Mechanisms with Infinite-State Stochastic Petri Nets. *Performance Evaluation Review*, 26(2):38–46, August 1998.

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

[23] W. R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC — 2001, January 1997.