An Investigation of Credit-based Flow Control Protocols[*]

Jonathan Billington
Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
jonathan.billington@unisa.edu.au

Smit Saboo
Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
smit.saboo@unisa.edu.au

ABSTRACT

Credit-based flow control mechanisms, such as those used in the Transmission Control Protocol, allow flow control and error control procedures of data transfer protocols to operate independently. We create a Coloured Petri Net model of a class of data transfer protocols, which uses retransmissions and acknowledgements for error control and "credits" for flow control. This model is characterized by 3 parameters: the maximum sequence number, the maximum number of retransmissions and the maximum receiver buffer size. From the analysis results, we derive expressions in these parameters for the channel bounds and the number of terminal states. These expressions are verified for a range of values of the parameters.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Verification, Experimentation

Keywords

Flow Control, Coloured Petri Nets, Exhaustive Simulation

1. INTRODUCTION

1.1 Background and Motivation

Flow control is an integral and an essential part of most data transfer protocols [16]. Without efficient flow control a fast sender can overwhelm a receiver. This may also lead to network congestion and bandwidth wastage. Flow control mechanisms are mainly employed in link layer and transport layer protocols [14]. Flow control is a rather complex mechanism at the transport layer, because it may have to deal with an unreliable underlying medium (such as the Internet Protocol) where the sending entity would not know whether the lack of an acknowledgement is due to flow control or packet loss. Moreover, if the receiver withholds the acknowledgement, to let itself process the data and free buffers for further incoming data, this withholding could lead to retransmission from the sender. Therefore flow control at the transport layer (such as TCP) is not linked with acknowledgements as it is in most data link protocols [16].

Stallings [14] calls this mechanism a credit scheme in which the receiver sends a credit (number of buffers available for buffering data) to the sender to let the sender know the receiver's condition and regulate its transmission rate. In this paper we aim to investigate this credit-based flow control mechanism by modelling it with Coloured Petri Nets (CPNs) [9]. We choose CPNs because of their clear graphical representation and their associated tools and techniques for analysis [2]. Moreover, CPNs have been successfully used to model and analyse communication protocols [8]. A model of the protocol specification is created and analysed using exhaustive simulation to characterize terminal states and communication channel bounds, in terms of the protocol's parameters.

1.2 Related Work

There have been several attempts to verify data transfer protocols. Chkliaev et al. [3] verified a sliding window protocol for window size 'n' and sequence number space '2n' using a theorem prover over an unreliable communication channel, which can re-order, lose and duplicate packets. Fokkink et al. [5] used axiomatic theory to verify the sliding window protocol again for any arbitrary finite window size 'n' but over a lossy queue of capacity one. In [12] unbounded sequence numbers are assumed when verifying a sliding window protocol for the transport layer. This simplifies verification because it is known that repetition (wrapping) of sequence numbers is the main source of errors for data transfer protocols [15]. The combination of abstraction techniques and model-checking allowed the sliding window protocol to be verified for a relatively large window size of 16, using the SPIN model checker [13]. Billington and Gallasch [1] showed how stop and wait protocols can fail over reordering channels and later verified a parameterised model of the class of Stop-and-wait protocols operating over lossy in-order channels by obtaining an explicit algebraic expression for the infinite set of state spaces as a function of the different parameters (retransmission limit and maximum se-

[*]This work was partially funded by ARC Discovery Grant, DP0559927.

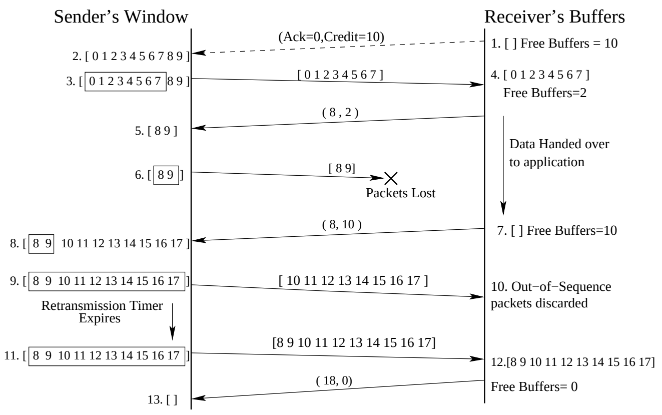

Figure 1: Credit-based flow control scenario

quence number) [6, 7].

All the work mentioned above uses acknowledgements for both flow control and error control. In contrast, as in the Transmission Control Protocol (TCP) [11], our protocol uses a credit-based scheme for flow control and retransmission (with acknowledgements) for error control. This decouples error control from flow control. We have not found any previous work that formally models and analyses this important class of data transfer protocols.

1.3 Contribution and Organisation

This paper provides two contributions:

- Firstly, we believe it is the first time that a class of credit-based flow control protocols is modelled and analysed using Coloured Petri Nets. This is important because credit-based flow control is what is used in real transport protocols (TCP) rather than sliding window protocols which have been analysed previously.

- Secondly, we derive expressions for the channel bounds and the number of terminal states for the CPN model, and validate them for a range of values of the protocol parameters.

The rest of this paper is organised as follows. Section 2 contains some background information about credit-based flow control. Section 3 discusses the assumptions made in our CPN model of the protocol, while the model is described in Section 4. Section 5 analyses the CPN model and some conclusions are drawn in Section 6. We assume familiarity with Coloured Petri Net terminology [9].

2. CREDIT BASED FLOW CONTROL

The principles behind credit-based flow control protocols are illustrated in Fig. 1. Each action in the time sequence diagram has been numbered and is explained below.

1. In Fig. 1, the Receiver allocates ten buffers and sends the first acknowledgement packet to the sender, stating, starting from sequence number '0' it has space for ten packets. For our purposes, we consider that an acknowledgement packet comprises a pair of two non-negative integers, (Ack,Credit). Ack is the sequence number (SN) of the packet that is next expected by the Receiver. Credit represents the number of free buffers in the Receiver. In this example, we assume that the receiver is expecting to receive a packet with SN=0 (hence Ack=0). Because 10 buffers have been allocated (and are free), Credit=10. This acknowledgement (shown by the dotted arrow) from the Receiver, usually occurs during connection establishment.

2. The Sender receives this acknowledgement and establishes its window, shown between square brackets. This window is at its maximum size representing the maximum number of buffers. The acknowledgement number determines the start of the window and the credit determines the end of the window. Packets with sequence number '0' to '9' are thus allowed to be transmitted.

3. Initially the Sender has eight packets ('0' to '7') to send and transmits them all. These packets are now called outstanding packets as they are waiting for acknowledgements from the receiver. These outstanding packets are enclosed in a rectangular box, which can be considered the retransmission queue.

4. The Receiver receives these packets and buffers them. It also sends an acknowledgement packet with acknowledgement number '8' and a credit of two, as eight out of ten buffers are occupied by received packets.

5. The Sender receives the acknowledgement and checks its window. The acknowledged outstanding packets are removed from its retransmission queue. A credit of 2 corresponds to the current window and hence it is not changed.

6. The Sender obtains another 10 packets from its user to send but can only transmit packets '8' and '9', which in this case are lost.

7. Meanwhile the Receiver delivers its packets to its application and in turn frees its buffers. It now has ten free buffers, which it advertizes to the sender by an acknowledgement.

8. The Sender receives the acknowledgement and increases its upper window limit to 17, since it is to send 10 packets starting from 8. It is still waiting for an acknowledgement for the outstanding packets '8' and '9'.

9. The Sender now transmits the new packets within the window, which are also placed in the retransmission queue along with packets '8' and '9'.

10. The Receiver receives these packets and discards them as they arrived out of order. Different accept policies can be used by the receiver, which we discuss below.

11. The retransmission timer for packet '8' times out and the whole retransmission queue is transmitted. There are different retransmission policies that the Sender may employ, and these are also discussed below.

12. The Receiver accepts all the in order packets and has all buffers occupied (Free buffers=0). It sends an acknowledgement with zero credit.

13. On receipt, the retransmission queue is cleared and the window is zero. The Sender cannot transmit any further packets until it receives some credit from the receiver.

2.1 Retransmission Policy

The packets which are transmitted by the sender are placed in a retransmission queue and can be retransmitted according to the following policies [14]:

- First-only: Only one retransmission timer is maintained for the entire queue. If an acknowledgement is received, the acknowledged packets are removed from the queue and the timer is reset. When the timer expires, the first packet is retransmitted, the timer is restarted and the retransmission count incremented by one.

- Batch: Again only one timer is maintained for the entire queue. When the timer expires, the whole queue is transmitted and the retransmission counter incremented.

Each of the above mentioned schemes has its pros and cons, moreover the efficiency of each of these policies depends upon the accept policy of the receiver, which is discussed below. The First-only retransmit policy is efficient in terms of traffic generated, because only the lost packets are retransmitted. Though this policy helps with congestion control, there can be considerable delays as the sender will always wait for the acknowledgement for the first packet, before it can retransmit any other packets.

In the batch retransmit policy, a single retransmission might take care of all the lost packets as the sender does not wait for the timer to timeout again for another retransmission. Although Batch retransmit reduces the likelihood of long delays, it performs badly in terms of congestion control. Retransmitting all packets on the queue when only one packet is lost (or its acknowledgement is lost) is very wasteful of bandwidth and may lead to congestion.

2.2 Receiver Accept Policy

According to [14], the receiver can implement one of the following accept policies:

- In-Order: The receiver only accepts in-order packets, discarding those that arrive out of order. Receivers implementing this policy are called non-buffering receivers.

- In-window: The receiver accepts all packets that are within the receiver's *receive window*, which depends on the number of free buffers. Receivers implementing this policy are usually called buffering receivers.

The in-order policy is easily implemented but it burdens the network, as the sender has to timeout and retransmit packets that were successfully received but discarded because of misordering. This accept policy is best suited to the Batch retransmit policy. The in-window accept policy reduces the burden on the network by reducing the number of retransmissions but requires a more complex acceptance test depending on the Receiver's window size and a more sophisticated data storage scheme to buffer and keep track of data accepted out of order. The In-window accept policy is more suited to the First only retransmit policy.

3. MODELLING ASSUMPTIONS

The assumptions made in modelling the credit scheme with CPNs are as follows:

1. Receiver and Sender are initially synchronized: Synchronization occurs during connection establishment so that the sender and receiver know the initial sequence number. The Sender also obtains the number of buffers the Receiver has allocated for the connection during connection establishment and adjusts its window accordingly. As it is part of connection establishment, synchronization is not included in the model. We assume the initial sequence number is zero for both the sender and the receiver. The number of buffers depends on a constant, MaxBufSize. MaxBufSize can take any positive integer value, but must be less than MaxSeqNo+1, to ensure no packets within the window have the same sequence number.

2. Data abstraction: The data transferred is not modelled, since the protocol behaves the same way irrespective of packet contents. Hence data packets are just represented by their sequence numbers.

3. Lossy and in-order communication channel: This corresponds to usual behaviour in many networks and is a good assumption to start with. Reordering channels result in more complex behaviour that will be the subject of future work.

4. Received data delivery: According to [14], a receiving entity is free to deliver the received data to the user at its own convenience. It can deliver the data to the user as soon as it receives an in-order packet or it can first buffer the data before delivery. The actual policy will depend on performance considerations. If the data transfer to the user is infrequent and large, there is low overhead and less processing, whereas if the data transfer is frequent and small, the system provides a quick response. This quick response is required mainly for interactive user applications. For non-interactive applications like file transfer, efficiency is more important than response time [16], hence buffering the received data until a large chunk of data is available for the user is more suitable for non-interactive applications. In this paper we assume a non-interactive application to be the user. Thus the receiver in the CPN model waits until all its buffers are full before transferring all the data to the user.

5. Limited Retransmission: We explicitly model the limit of retransmission (MaxRetrans), as these limits exist in practice. Thus the system will terminate when all the retransmissions are lost (or the acks are lost). In practice, when the sender reaches its limit (MaxRetrans), and no acknowledgement is received within a certain period, the user is notified of this condition and the connection (or session) is cleared down. However, as we are only modelling the flow control procedures, we do not model this closing procedure. Hence we expect to obtain a set of terminal states that relate to this condition in the sender.

6. Acknowledgements of received data packets: The receiver can receive any number of packets, before they are acknowledged. This makes our CPN model more general and also considers the more usual case of acknowledging every packet received.

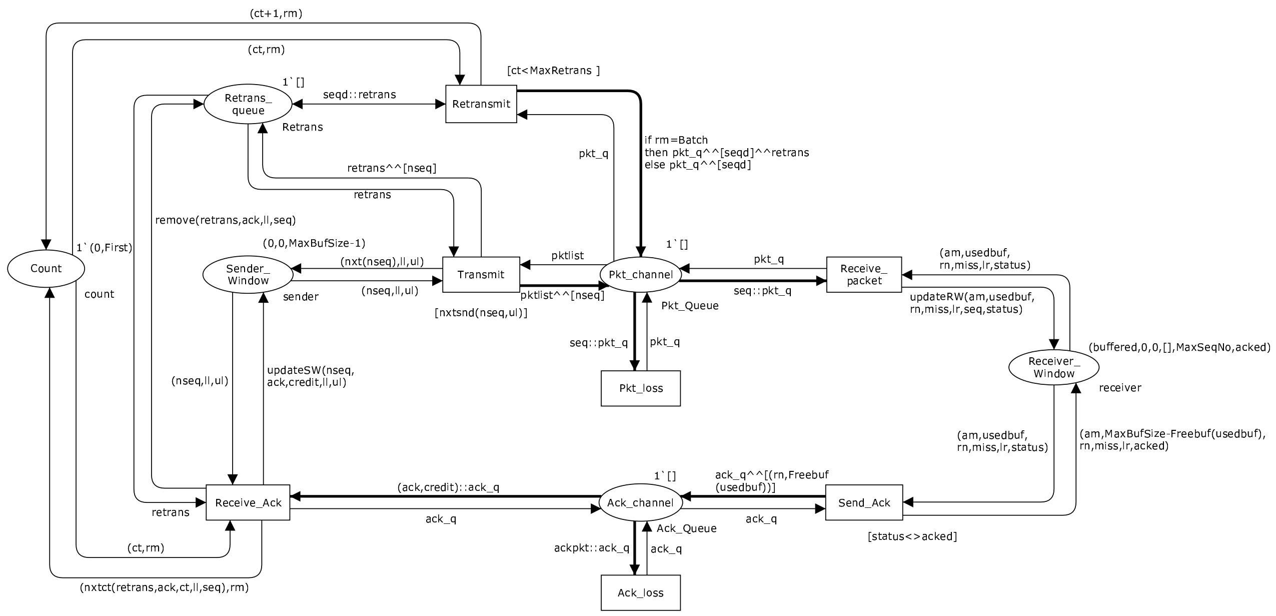

Figure 2: CPN of Credit-based Flow Control Protocol

```
1  (* Parameters *)
2  val MaxSeqNo= 7;
3  val MaxBufSize= 5;
4  val MaxRetrans= 1;

5  (* Colour Sets associated with Sender *)
6  colset Seq = int with 0..MaxSeqNo;
7  colset RM= with First | Batch;
8  colset count= product INT *RM;
9  colset Retrans= list Seq;
10 colset sender= product Seq*Seq*Seq;

11 (* Colour Sets associated with Receiver *)
12 colset Status= with acked | notacked;
13 colset AM= with order | buffered;
14 colset Buffers= int with 0..MaxBufSize;
15 colset Missing=list Seq;
16 colset receiver= product AM*Buffers*Seq*Missing*Seq*Status;

17 (* Colour Sets associated with Communication Channel *)
18 colset Pkt_Queue= list Seq;
19 colset Ack= product Seq*Buffers;
20 colset Ack_Queue=list Ack;

21 (* Variables *)
22 var seq,nseq,seqd,rn,lr,ll,ul,ack: Seq;
23 var rm: RM;
24 var ct:INT;
25 var retrans: Retrans;
26 var status: Status;
27 var am: AM;
28 var usedbuf,credit:Buffers;
29 var miss: Missing;
30 var pkt_q: Pkt_Queue;
31 var ackpkt: Ack;
32 var ack_q: Ack_Queue;
```

Figure 3: Colour Set declaration for the CPN model

4. THE CPN MODEL

With the assumptions above, a CPN model was developed using the software package CPN Tools [4]. The CPN model is given in Fig. 2 and Fig. 3 lists the colour sets and variables for our CPN model. The model consists of a Sender communicating with a Receiver over lossy FIFO (First-in, First-out) channels.

4.1 Sender

The sender part consists of three places: `Sender_Window`, `Retrans_queue` and `Count` and three transitions: `Transmit`, `Retransmit` and `Receive_Ack`. Each of the places and transitions are discussed below.

4.1.1 Places

The place `Sender_Window` represents the sender's window. It is typed by the colour set, sender, which is the product of three sequence numbers (line 10, Fig. 3). Colour set, Seq (line 6, Fig. 3), defines the set of sequence numbers as a finite range of integers from zero to the maximun sequence number (MaxSeqNo). A token on place `Sender_Window` is of the form (nseq,ll,ul), where :

- nseq: represents the sequence number of the next packet to be transmitted.
- ll represents the lower limit of the window.
- ul represents the upper limit of the window.

The initial marking (state) of place `Sender_Window` is (0,0, MaxBufSize-1), which means the first sequence number to be transmitted is zero, the window starts from sequence number zero and the upper limit is given by MaxBufSize minus one. Thus up to the window of packets can be sent initially.

After a packet is transmitted it is placed on a retransmission queue represented by place `Retrans_queue`. Packets remain in this queue until they have been acknowledged. The place is typed by Retrans (line 9) which is a list of sequence numbers. Initially the retransmission queue is empty, represented by the empty list [ ].

Place `Count` represents the retransmission counter and the retransmission mode. The counter is an integer, representing the number of times the retransmission timer expires for a particular packet. The counter can never exceed the maximum retransmission value (MaxRetrans). The retransmission mode, RM (line 7), indicates which retransmission policy the sender is employing (either First_only or Batch). The place `Count` is thus typed by the product of an integer and RM (line 8). The initial marking of `Count` indicates which retransmission mode is used, e.g. for the Batch-retransmit policy, the initial marking is (0,Batch).

4.1.2 *Transitions*

When `Transmit` occurs the packet to be transmitted (nseq from place Sender) is appended to the channel list of packets and the next sequence number is incremented modulo MaxSeqNo+1 by using function `nxt`. The guard of `Transmit` is the function `nxtsnd(nseq,ul)` which ensures that the transmitted packet is not outside the window of the sender.

`Receive_Ack` processes acknowledgements received by the sender. Acknowledgments comprise an acknowledgement number (ack) and the amount of credit (number of free buffers) available at the receiver. Firstly, it determines if the ack is valid or not. An ack is valid if it falls in the range [ll,nseq] (taking sequence number wrap modulo (MaxSeqNo+ 1) into account). If the ack is not valid, then the acknowledgement packet is removed from the channel and discarded. When ack=ll, the acknowledgement is a duplicate. In this case, it is the credit value that is important, and is used to calculate the new value of the upper limit of the window (ul). This is done by setting ul=(ack+(credit−1)) mod (MaxSeqNo + 1), which is implemented in `updateSW(nseq,ack,credit,ll,ul)`. No other action is taken apart from removing the packet from the channel. The final case is for ack in the range [ll+1,nseq], when packets are acknowledged. The acknowledged packets (those in the range [ll, ack−1]) are removed from the head of the retransmission queue, the upper limit of the window is also updated according to the credit received (as indicated in the previous case) and the retransmission counter is reset to zero. These actions are implemented using the `remove`, `updateSW` and `nxtct` functions respectively.

`Retransmit` models retransmission differently for the two retransmission modes. When the sender is using the First-only policy (rm=first), only the first item on the retransmission queue is appended to the list of packets in the channel, however, for the Batch policy (rm=batch), the whole of the retransmission queue is appended. It also increments the retransmission counter by one. The inscription on the double arc between `Retrans_queue` and `Retransmit` has been chosen to be "seqd::retrans" (with the corresponding if-then-else statement on the arc leading to the channel) to ensure that retransmission does not occur if the queue is empty. The number of retransmissions is limited to MaxRetrans by the guard.

4.2 Communication Channel

Communication channels between the Sender and Receiver (`Pkt_channel` and `Ack_channel`) are implemented as lossy FIFO queues in the usual way. The channel contents are treated as a list of packets (or acknowledgements), the concatenation operator (^^) is used to add packets (or a list of packets corresponding to the retransmission buffer) to the end of the current queue and the cons operator (::) allows the first item of the queue to be served. Without loss of generality, loss of packets is considered to occur from the front of the queue, and is implemented by the transitions `Pkt_loss` (for packets) and `Ack_loss` (for acknowledgements). Packets are just represented by their sequence number (the data is irrelevant) whereas acknowledgements are modelled as a pair (see line 19, Fig. 3) comprising the acknowledgement number (which is the next expected sequence number) and the credit corresponding to the number of free buffers in the receiver. Initially both channels are empty represented by an empty list [].

4.3 Receiver

The model of the receiver would be simple if we just implemented the in-order acceptance policy, i.e. a non-buffering receiver. However, we are interested in the behaviour of flow control protocols where a buffering receiver may also be implemented. We therefore chose to model both behaviours. To do this we define two acceptance modes that we call order (in-order policy) and buffered (in-window policy). These are defined in the AM set in the declarations (see line 13, Fig. 3) The particular policy is chosen via the CPN model's initial marking. To cater for the buffering receiver we need to define some data structures which allow us to characterise and manage the buffers. Firstly we define the parameter MaxBufSize to represent the number of buffers that the operating system has allocated to the receiver. We consider that this number is fixed for the duration of the connection. Now we define the following data structures:

- Buffers (line 14) represents the number of buffers that are currently being used;
- Missing (line 15) is a list of sequence numbers of all the packets that are required to fill any gaps in the receiver's buffers. For example, if the receiver has stored packets 1, 3 and 6 in its buffers then the missing packets would be the list [2,4,5]; and
- Status (line 12) indicates whether the last received packet has been acknowledged or not.

The type of the place `Receiver_Window` (representing the state of the Receiver) is determined as the product of 6 sets: AM, Buffers, Seq, Missing, Seq and Status (line 16). The third component represents the next expected sequence number from the sender, whereas the 5th component represents the sequence number of the last received packet. The initial marking for a non-buffering receiver would be (order,0,0,[],MaxSeqNo,acked), where order indicates a non-buffering mode; all the buffers are available; the next expected sequence number is 0; there are no missing packets in the buffer (because it is empty); given that the next expected packet has sequence number 0, then the last packet received is initialised to MaxSeqNo (the previous sequence number); and finally acked indicates that the last received packet has been acknowledged, so that the receiver does not send a gratuitous acknowledgement initially. The initial marking of place `Receiver_Window` in Fig. 2 is for a buffering receiver.

4.3.1 *Transitions*

Transition `Receive_packet` models receiving of data packets by the receiver. Its operation depends on the accept policy of the receiver. For an in-order policy, `Receive_packet` accepts only in-order packets and discards all out of order packets. For the in-window accept policy, all the packets that are within the receive window need to be buffered. This operation is complex and requires variables to store information about missing packets and out of order buffered packets. Transition `Receive_packets` uses the function `updateRW` to implement both policies.

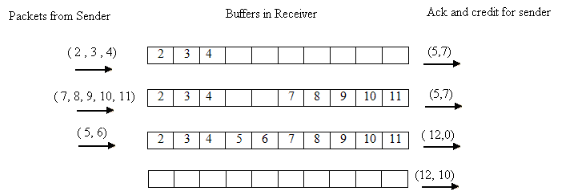

Figure 4: Buffering process in the Receiver

In-order Accept policy.

Transition `Receive_packet` calls function `updateRW(am, usedbuf,rn,lost,lr,seq,status)` on its output arc to the place `Receiver_Window`. If am=order, it checks if the received packet is in order (seq=rn) and also if there are free buffers available at the receiver to store the received packets (usedbuf < MaxBufSize). When both the conditions are satisfied, it increments the value of usedbuf and rn by one and updates the value of the last received packet to seq. It sets status to notacked, allowing the receiver to send an acknowledgement, if it wants to. If the packet arrived out of order (seq<>rn) or there were no buffers available then the state of the receiver remains unchanged,except that status is set to notacked, so that the receiver can send an acknowledgement.

In-window Accept policy.

Before going into the implementation of the in-window accept policy, we firstly discuss its principles in more detail. When an out-of-order packet arrives, which is still within the *receive window*, it is stored in the buffer leaving enough buffer space for the missing packets. When the missing packets arrive, they are stored in the reserved buffers. When all the buffers are occupied, all the data is delivered to the user, freeing up all the buffers to receive further packets.

Figure 4 illustrates the buffering process where the receiver has allocated ten buffers for the connection. These ten buffers form the receive window. The sequence number which the receiver is expecting, plus the next nine sequence numbers, are within the receive window and can be buffered, if they arrive out of sequence. When the first three packets arrive (in sequence), they are stored in the first three buffers and an acknowledgment is sent along with the credit. The acknowledgement indicates that all packets have been received up to and including the packet with sequence number 4 and the credit is for seven buffers (as three out of ten buffers are occupied). When the next packet arrives, its sequence number is out of sequence. The receiver learns that two packets are missing and stores the received packet after reserving two buffers for missing packets. The receiver does not count out-of-order buffered packets as occupying buffers. Therefore after receiving the out of order packets (see Fig. 4) the number of used buffers is still considered to be three. Therefore the receiver sends a packet with acknowledgement number 5 and credit 7, even though only two buffers are unoccupied. Now the missing packets (5 and 6) arrive, and the receiver stores them in their reserved buffers. It has now received packets with sequence number (up to and including) '11' in sequence and all its buffers are full, so it sends an acknowledgement (12) with zero credit. Once all the buffers are full, all the data is delivered to the user and the buffers are available, so the receiver sends a new credit of 10 buffers.

Transition `Receive_packet` uses variables miss and lr to implement the in-window accept policy. The buffering receiver procedures are implemented when the variable am is bound to 'buffered', which will be the case for Fig. 2. When am=buffered, function `updateRW` first checks if the received sequence number is the expected sequence number (rn) or one of the sequence numbers which it did not receive earlier, i.e. is in miss. If either of these two conditions is true and it still has free buffers (usedbuf < MaxBufSize) then it changes the state of the receiver in the following way:

- `updateRW` uses `total(usedbuf,lost,seq,lr)` to calculate the total number of used buffers. If the received packet is expected (seq=rn) and miss is empty then it simply increments usedbuf by one. If the packet is one of the missing ones then it uses the following logic to calculate usedbuf:

    1. If the received packet sequence number is at the head of the missing packet list (miss), increment usedbuf by the difference between the received packet sequence number and the second packet of the list.
    2. If the missing list has just one packet which is equal to the received packet, increment usedbuf by the difference between the sequence number of the received packet and the last out of sequence received packet (lr).
    3. If the received packet is not the head of the list, do not update usedbuf as there are still missing packets ahead of it.

- The received packet is removed from the missing packets list.
- If miss is the empty list, the last received number is set to the received sequence number, otherwise it is not changed.
- When a packet is received the status is changed from acked to notacked.

If the received sequence number is not the expected sequence number and it is not among the missing numbers, `updateRW` checks if the sequence number is within the receive window by calling `valid(rn,seq,MaxBufSize−usedbuf)`. If in range, it changes the state of the receiver as follows:

- Since an out of sequence packet is buffered, there are some packets missing. `updateRW` calls `add(rn,seq,lr, miss,usedbuf)` which calculates the missing sequence numbers and appends them to the current missing packet list.
- The last received sequence number is updated to the new received sequence number.
- When a packet is received the status is changed from acked to notacked.
- None of the other receiver state variables are changed.
- Finally, if the received sequence number is not in the receive window, the packet is discarded.

`Send_Ack` models the generation of acknowledgements as well as credit for the sender. When it occurs, it creates an acknowledgement with 'rn' as the acknowledgement number and the difference between `MaxBufSize` and `usedbuf` (`MaxBufSize-usedbuf`) as the credit. The pair is appended to the list of acknowledgements in the Ack_channel. Our assumption that the receiver waits for its buffers to be full before it delivers data to the user is implemented in this transition. When usedbuf is equal to MaxBufSize (i.e. all the allocated buffers are now occupied), usedbuf is set to zero and credit, equal to MaxBufSize, is sent to the sender. The transition's guard [status = notacked] ensures the receiver only sends acknowledgements after it receives a data packet.

5. ANALYSIS OF OCCURRENCE GRAPHS

Exhaustive simulation of the CPN model was performed using CPN tools [4, 10]. This was done by generating the set of all states and state changes known as the Occurrence Graph (OG), for a range of parameter values, as the OGs depend on three model parameters: MaxBufSize, MaxSeqNo and MaxRetrans.

Table 1 shows the results obtained for a set of OGs for the sender having a batch-retransmit policy and the receiver having in-order reception. The OGs were generated having a lossy FIFO communication channel over a range of values of the parameters.

Since parameter MaxBufSize determines the initial window size at the Sender, we use W (window) as a shorthand for MaxBufSize. In Table 1, the first three columns record the values of the parameters (R for MaxRetrans, S for MaxSeqNo and W (window) for MaxBufSize). The next three columns give the number of nodes, arcs and dead markings (DMs) in each OG. The last column indicates the bounds for each communication channel (CB).

5.1 Number of Dead Markings

In the following we derive the number of dead markings of the CPN model, for an In-order Receiver, as a function of MaxBufSize and MaxSeqNo. A dead marking is a terminal state of the model in which there is no state change. Dead markings for the CPN model of Fig. 2 are those in which all 7 transitions of the model are not enabled. The following summarizes the conditions under which all 7 transitions will not be enabled.

- `Transmit` is not enabled only when its guard, nxtsnd(nseq, ul), is false because in every marking in $[M_0\rangle$ (the set of reachable markings, from the initial marking M_0)), the input arc expressions for `Transmit` will be satisfied by the places `Sender_Window`, `Retrans_queue` and `Pkt_channel`. The variables nseq and ul take their values from the marking of `Sender_Window`. Thus all the reachable markings of `Sender_Window` satisfying nxtsnd(nseq,ul)=false will contribute to dead markings if all the other transitions are not enabled. Note that nxtsnd(nseq,ul)=false implies that nseq= nxt(ul) (i.e. the next packet to send is outside the window).

- `Retransmit` is not enabled when "ct=MaxRetrans" due to the guard expression. `Retransmit` is also not enabled when M(Retrans_queue)= 1`[ ] but M(Retrans_queue) =1`[ ] will never be true in a dead marking. If

R	S	W	Nodes	Arcs	DMs	CB
0	1	1	12	12	4	(1,1)
0	2	1	18	18	6	(1,1)
0	3	1	24	24	8	(1,1)
0	4	1	30	30	10	(1,1)
0	5	1	36	36	12	(1,1)
0	6	1	42	42	14	(1,1)
0	7	1	48	48	16	(1,1)
0	8	1	54	54	18	(1,1)
0	9	1	60	60	20	(1,1)
1	1	1	82	206	4	(3,3)
1	2	1	123	309	6	(3,3)
1	3	1	164	412	8	(3,3)
1	4	1	205	515	10	(3,3)
1	5	1	246	618	12	(3,3)
1	6	1	287	721	14	(3,3)
1	7	1	328	824	16	(3,3)
1	8	1	369	927	18	(3,3)
1	9	1	410	1030	20	(3,3)
2	1	1	270	902	4	(5,5)
2	2	1	405	1353	6	(5,5)
2	3	1	540	1804	8	(5,5)
2	4	1	675	2255	10	(5,5)
2	5	1	810	2706	12	(5,5)
2	6	1	945	3157	14	(5,5)
2	7	1	1080	3608	16	(5,5)
2	8	1	1215	4059	18	(5,5)
2	9	1	1350	4510	20	(5,5)
0	2	2	84	132	15	(2,2)
0	3	2	56	88	10	(2,2)
0	4	2	140	220	25	(2,2)
0	5	2	84	132	15	(2,2)
0	6	2	196	308	35	(2,2)
0	7	2	112	176	20	(2,2)
0	8	2	252	396	45	(2,2)
0	9	2	140	220	25	(2,2)
1	2	2	1467	5202	15	(7,7)
1	3	2	978	3468	10	(7,7)
1	4	2	2445	8670	25	(7,7)
1	5	2	1467	5202	15	(7,7)
1	6	2	3423	12138	35	(7,7)
1	7	2	1956	6936	20	(7,7)
1	8	2	4401	15606	45	(7,7)
1	9	2	2445	8670	25	(7,7)
2	2	2	7557	31380	15	(12,12)
2	3	2	5038	20920	10	(12,12)
2	4	2	12595	52300	25	(12,12)
2	5	2	7557	31380	15	(12,12)
2	6	2	17633	73220	35	(12,12)
2	7	2	10076	41840	20	(12,12)
2	8	2	22671	94140	45	(12,12)
2	9	2	12595	52300	25	(12,12)
0	3	3	360	756	36	(3,3)
0	4	3	450	945	45	(3,3)
0	5	3	180	378	18	(3,3)
0	6	3	630	1323	63	(3,3)
0	7	3	720	1512	72	(3,3)
0	8	3	270	567	27	(3,3)
0	9	3	900	1890	90	(3,3)
0	10	3	990	2079	99	(3,3)
0	11	3	360	756	36	(3,3)
1	3	3	14376	57124	36	(12,12)
1	4	3	17970	71405	45	(12,12)
1	5	3	7188	28562	18	(12,12)
1	6	3	25158	99967	63	(12,12)
1	7	3	28752	114248	72	(12,12)
1	8	3	10782	42843	27	(12,12)
1	9	3	35940	142810	90	(12,12)
1	10	3	39534	157091	99	(12,12)
1	11	3	14376	57124	36	(12,12)

Table 1: OGs for Batch-retransmit sender and in-order receiver for a range of parameter values.

all the outstanding packets are acknowledged, nseq=ll and as ul≥ll, seq≠nxt(ul) and `Transmit` will be enabled. Hence there will be only one marking of place `Count` (1`(MaxRetrans, First) or 1`(MaxRetrans, Batch)) depending on the initial marking) that contributes to dead markings.

- `Receive_Ack` and `Ack_loss` are not enabled only when M(Ack_Channel) = 1`[ ]. Hence the `Ack_channel` is empty in all dead markings.
- Likewise `Pkt_channel` is empty in all dead markings.
- Finally, `Send_Ack` is not enabled only when "status = acked".

For each reachable marking of `Sender_Window` in which "seq =nxt(ul)" (and the other places have markings as discussed above), all reachable markings for `Receiver_Window` in which the last component of the token is acked, will give the number of dead markings.

The initial marking of `Sender_Window` depends upon the parameter, MaxBufSize. For MaxBufSize=W, the window will be W and `Sender_Window` will have W reachable markings in which condition "nseq=nxt(ul)" is satisfied. For W=4, we have 4 such reachable markings

- Sender_Window$_0$(S_W$_0$): The sender has transmitted and retransmitted all the packets within the window and has received no acknowledgements for packets within the window, so that ("nxtsnd(nseq,ul)=false").
- Sender_Window$_1$(S_W$_1$): The sender has transmitted and retransmitted all the packets within the window and has received an acknowledgement for the first transmitted packet.
- Sender_Window$_2$: The sender has transmitted and retransmitted all the packets within the window and has received acknowledgements for the first two transmitted packets.
- Sender_Window$_{(W-1)}$(S_W$_{(W-1)}$) or Sender_Window$_3$ (S_W$_3$): The sender has transmitted and retransmitted all the packets within the window and has not received an acknowledgement for the last transmitted packet.

Now for each reachable marking of `Sender_Window` (which has `Transmit` not enabled) we have the following corresponding reachable markings for `Receiver_Window` in which `Send_Ack` is not enabled.

The corresponding reachable markings of `Receiver_Window` for marking Sender_Window$_0$; which have status=acked are given by:

1. Receiver_Window$_{00}$(R_W$_{00}$): All the transmitted and the corresponding retransmitted packets were lost so no packets were received.
2. Receiver_Window$_{01}$(R_W$_{01}$): The first packet was received and all other packets were lost; Receiver sents an acknowledgement which was also lost.
3. Receiver_Window$_{02}$(R_W$_{02}$): The first two packets were received and the last two packets were lost; Receiver sents acknowledgements for the first two packets, which were also lost.
4. Receiver_Window$_{0(W-1)}$(R_W$_{0(W-1)}$) or Receiver_Window$_{03}$(R_W$_{03}$): The first three packets were received and the last packet was lost; Receiver sents acknowledgements, which again were also lost.
5. Receiver_Window$_{0W}$(R_W$_{0W}$) or Receiver_Window$_{04}$ (R_W$_{04}$): All packets were received but all the acknowledgements were lost.

Hence, for reachable marking Sender_Window$_0$ we have W+1 or 5 reachable markings for place `Receiver_Window`. Using similar arguments, we obtain the Table 2 of reachable markings for `Receive_Window` for each of the `Sender_Window` markings.

M(Sender_Window)	M(Receiver_Window)	Number of Markings
S_W$_0$	{R_W$_{0i}$| i ∈ {0,w}}	W+1
S_W$_1$	{R_W$_{1i}$| i ∈ {0,w-1}}	W
...	...	...
S_W$_{W-1}$	{R_W$_{(W-1)i}$| i ∈ {0,1}}	2

Table 2: Number of Dead Markings

The total number of dead markings is therefore given by the sum of all the reachable markings of `Receive_Window`:

$$2 + \ldots + (W-1) + W + (W+1) \tag{1}$$

$$= (1 + 2 + \ldots + (W-1) + W) + W \tag{2}$$

$$= \frac{W(W+1)}{2} + W = \frac{W(W+3)}{2} \tag{3}$$

Equation(3) gives the dead markings only for the first window. Because of the credit mechanism, the window at the Sender is MaxBufSize. Because of the delivery policy at the receiver (Assumption 4), the window at the sender moves across the sequence numbers in discrete steps of size equal to the MaxBufSize (W). The window moves m-number of times over n-sequence number spaces (sequence number space={0,1...MaxSeqNo}) before the states of Sender_Window start repeating. For example, for W=2 and sequence number space=3 (MaxSeqNo=2), m=3 and n=2. This is shown in Fig. 5. In Fig. 5, the window moves three times

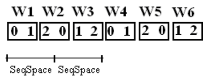

Figure 5: Movement of the window over several sequence number spaces

before the sender's states start repeating. The starting and ending sequence numbers of W1, W2 and W3 (W1...W6 are here called window state classes) are equal to the starting and ending sequence numbers of W4, W5 and W6 respectively. Also W1, W2 and W3 run over two sequence number spaces together, therefore in Fig. 5, m=3 and n=2.

We define the *Window State Class* (WSC), which records the number of different window state classes (m) that occurs as the window runs over n sequence number spaces. WSC may also be defined as the number of different window state classes which are possible for a given sequence number space (SS) and MaxBufSize (W). Table 3 gives the statistics for WSC for different values of MaxSeqNo (S) and MaxBufSize. Some interesting trends appear in Table 3. The number of state classes for the sender, WSC, is obtained by dividing the

W	S	SS	n	WSC	W	S	SS	n	WSC
2	2	3	2	3	4	14	15	4	15
2	3	4	1	2	4	15	16	1	4
2	4	5	2	5	5	5	6	5	6
2	5	6	1	3	5	6	7	5	7
2	6	7	2	7	5	7	8	5	8
2	7	8	1	4	5	8	9	5	9
2	8	9	2	9	5	9	10	1	2
2	9	10	1	5	5	10	11	5	11
3	3	4	3	4	5	11	12	5	12
3	4	5	3	5	5	12	13	5	13
3	5	6	1	2	5	13	14	5	14
3	6	7	3	7	5	14	15	1	3
3	7	8	3	8	5	15	16	5	16
3	8	9	1	3	6	6	7	6	7
3	9	10	3	10	6	7	8	3	4
3	10	11	3	11	6	8	9	2	3
3	11	12	1	4	6	9	10	3	5
4	4	5	4	5	6	10	11	6	11
4	5	6	2	3	6	11	12	1	2
4	6	7	4	7	6	12	13	6	13
4	7	8	1	2	6	13	14	3	7
4	8	9	4	9	6	14	15	6	15
4	9	10	2	5	6	15	16	3	8
4	10	11	4	11	6	16	17	6	17
4	11	12	1	3	6	17	18	1	3
4	12	13	4	13	6	18	19	6	19
4	13	14	2	7	6	19	20	3	10

Table 3: WSC for a range of values of MaxBufSize and MaxSeqNo

sequence number space by the greatest common divisor of the sequence number space and MaxBufSize; and n is given by the quotient: MaxBufSize divided by the same greatest common divisor. Therefore WSC is given by

$$WSC_{WS} = \frac{MaxSeqNo + 1}{\gcd(MaxBufSize, MaxSeqNo + 1)} \qquad (4)$$

where gcd finds the greatest common divisor of its two arguments. WSC$_{WS}$ is defined as the number of sender window state classes which are possible for a given maximum sequence number (S) and maximum window size (W) which corresponds to MaxBufSize. As equation (3) gives the dead markings for the first window, the total number of dead markings would be the sum of dead markings in every window. As the number of markings in each state class is the same the total number of dead markings for the In-order receiver is:

$$DM(In - O)_{WS} = WSC_{WS} \times \left(\frac{W(W+3)}{2} \right) \qquad (5)$$

Using similar arguments, we derived the following expression for number of dead markings for the In-window receiver.

$$DM(In-W)_{WS} = WSC_{WS} \times (2W + \sum_{i=1}^{W-1} (W-i) \times 2^i) \quad (6)$$

The results obtained from equation (5) are consistent with those provided in Table 1.

5.2 Channel Bounds

We now consider the maximum number of packets that can be in either the Packet Channel or Acknowledgement Channel for the Batch retransmit policy.

The bounds on the number of packets in both the forward and reverse channels are given in Table 1 for small values of the protocol's parameters when the sender implements the Batch retransmit policy. If we examine the first marking of place `Pkt_Channel`, when the list is at its maximum for the case when MaxRetrans=2, MaxBufSize=2 and MaxSeqNo=7, we obtain the following result (assuming the first packet to be transmitted has SN=0):

$$M(Pkt\_channel)_{max} = 1`[0\ 1\ 0\ 1\ 1\ 1\ 2\ 3\ 2\ 3\ 2\ 3]$$

Examining the structure of this result we can see that the first full window (01) has been retransmitted twice, then (1) has been retransmitted twice, given the successful receipt and acknowledgement of (01). This then allows the next window (23) to be transmitted and then retransmitted twice (MaxRetrans=2). At this stage, no further packets can be transmitted.

If we now chose MaxRetrans=2, MaxBufsize=3 and MaxSeqNo=7, we obtain:

$$M(Pkt\_channel)_{max} = 1`[0\ 1\ 2\ 0\ 1\ 2\ 1\ 2\ 1\ 2\ 2\ 2\ 3\ 4\ 5\ 3\ 4\ 5\ 3\ 4\ 5]$$

We can see that the first full window of packets (012) has been retransmitted twice, then one less than the full window (12) has been retransmitted twice, then the last packet of the window (2) has been retransmitted twice. The first transmission of (012) has been successfully received and acknowledged, allowing the next window of packets (345) to be transmitted and then retransmitted twice (the limit of retransmission). Given this structure, and noticing that there can only be packets in the channel from two consecutive windows (due to the flow control mechanism) we can now generalise this result as follows. Let R = MaxRetrans, and W=MaxBufSize, then

$$\begin{aligned} &MaxPackets(Batch)\_PC_{R,W} \\ &\quad = R(W + \ldots + 2 + 1) + (R+1)W \\ &\quad = \frac{R(W+1)W}{2} + (R+1)W \\ &\quad = \frac{1}{2}RW^2 + \frac{1}{2}(3R+2)W \qquad (7) \end{aligned}$$

Since the receiver can receive and acknowledge packets one at a time, all the packets in the `Pkt_channel` can be converted into their corresponding acknowledgements in the `Ack_channel`. Thus the above result also applies to the `Ack_channel`. These results have been confirmed for the range of parameter settings given in Table 1. Based on similar arguments the bound on both communication channels for the First-only retransmit policy is given by:

$$MaxPackets(First)\_PC_{R,W} = (R+1)W + R \qquad (8)$$

The channel bounds given in equations (7) and (8) are independent of the Receiver's accept policy. These results show that the channel bound is quadratic in MaxBufSize, when the Batch retransmit policy is used, and linear when the First-only retransmit policy is used. Both are linear in the maximum number of retransmissions.

5.2.1 *Comparison with the Stop-and-Wait Protocol*

When MaxBufSize= 1, the results presented here should be the same as those for the Stop-and-Wait protocol investigated in [7] where it was proved that the channel bounds were given by 2MaxRetrans + 1. If we substitute W=1 in equations (7) and (8) we obtain the same results:

$MaxPackets(Batch)\_PC_{R,1} = (4R+2)/2 = 2R+1$
and
$MaxPackets(First)\_PC_{R,1} = R+1+R = 2R+1$

6. CONCLUSIONS AND FUTURE WORK

This paper has modelled, for the first time, a class of credit-based flow control data transfer protocols operating over a lossy in-order medium using Coloured Petri Nets. The model is quite general and incorporates 3 main protocol parameters (maximum number of retransmissions (R), maximum sequence number (S) and maximum credit (W)) and policies regarding the behaviour of the sender (2 retransmission policies) and receiver (two acceptance policies). The model was analysed using exhaustive simulation for a range of parameter and policy settings. From these results we were able to obtain general expressions for the number of terminal states of the protocol as a function of two parameters: maximum sequence number and maximum credit (which also corresponds to the maximum number of buffered packets in the receiver and the maximum window size for packet transmission in the sender). Different expressions were derived for the two different acceptance policies used by the receiver, both of which demonstrated that the number of terminal states is independent of the retransmission limit. In order to obtain these expressions we have defined the notion of the number of window state classes. This depends on the size of the sequence number space and greatest common divisor of the sequence number space and the maximum credit. We believe it is the first time that this observation has been made.

Further we have obtained two different general expressions for the channel bounds for the both the batch and first only retransmit policies. These expressions only involve two parameters: maximum buffer size and maximum number of retransmissions. These expressions show that the bounds are quadratic in W for batch retransmission, while linear in W for first only retransmission, while both are linear in R. These results are validated using multiple exhaustive simulations for small values of the parameters. The channel bounds are also compared with those obtained for the simpler Stop-and-Wait protocol class and are shown to be consistent.

In the future we would like to prove the results for the number of terminal states and the channel bounds, and derive expressions in this protocol's parameters for the whole state space. This paper thus derives preliminary results that provide insight into the much more difficult task of obtaining expressions for the infinite family of state spaces in the three parameters. We would also like to explore the effect of reordering channels on these results, starting with the Stop-and-Wait class (W=1).

7. ACKNOWLEDGMENTS

The authors thank their colleagues, Guy Gallasch and Nimrod Lilith for their valuable suggestions and the referees for their constructive comments that have helped to improve the quality of this paper.

8. REFERENCES

[1] J. Billington and G. E. Gallasch. How Stop and Wait protocols Can Fail Over the Internet. In *Proceedings of FORTE'03, volume 2767 of Lecture Notes in Computer Science*, volume 2000, pages 209–223. Springer-Verlag, 2003.

[2] J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verication. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets, volume 3098 of Lecture Notes in Computer Science*, pages 210–290. Springer-Verlag, 2004.

[3] D. Chkliaev, J. Hooman, and E. Vink. Verification and Improvement of the Sliding Window Protocol. In *Tools and Algorithms for the Construction and Analysis of Systems, volume 2619 of Lecture Notes in Computer Science*, pages 148–163. Springer-Verlag, 2004.

[4] CPN Tools Online. `http://www.daimi.au.dk/CPNTools/`.

[5] W. Fokkink, J. F. Groote, J. Pang, B. Badban, and J. Pol. Verifying a Sliding Window Protocol in μCRL. In *Tools and Algorithms for the Construction and Analysis of Systems, volume 2619 of Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2003.

[6] G. E. Gallasch and J. Billington. Using Parametric Automata for the Verification of the Stop-and-Wait Class of Protocols. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005), Taipei, Taiwan, Volume 3707 of Lecture Notes in Computer Science*, pages 457–473. Springer.

[7] G. E. Gallasch and J. Billington. A Parametric State Space for the Analysis of the Infinite Class of Stop-and-Wait Protocols. In *13th International SPIN Workshop on Model Checking of Software, volume 3925 of Lecture Notes in Computer Science*. Springer, May 2006.

[8] J.Billington, M.Diaz, and G.Rozenberg, editors. *Application of Petri Nets to Communication Networks*. volume 1605 of Lecture Notes in Computer Science. Springer, 1999.

[9] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1-3. Springer, 1997.

[10] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, June 2007. Available via `http://dx.doi.org/10.1007/s10009-007-0038-x`.

[11] J. Postel. *Transmission Control Protocol*. IETF, September 1981.

[12] M. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Formal methods for distributed system development*, pages 19–34. Kluwer Academic Publishers, 2000.

[13] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, Abstract and Model-Check. In *Theoretical and Practical Aspects of SPIN Model Checking, volume 1680 of Lecture Notes in Computer Science*, pages 57–76. Springer-Verlag, 1999.

[14] W. Stallings. *Data and Computer Communications*. Prentice Hall, 8 edition, 2007.

[15] N. Stenning. A Data Transfer Protocol. *Computer Networks 1*, pages 99–110, 1976.

[16] A. Tanenbaum. *Computer Networks*. Prentice Hall, 4 edition, 2003.