Discovering Chatter and Incompleteness in the Datagram Congestion Control Protocol

Somsak Vanit-Anunchai[1], Jonathan Billington[1], and Tul Kongprakaiwoot[2]

[1] Computer Systems Engineering Centre, University of South Australia, Mawson Lakes Campus, SA 5095, Australia
Somsak.Vanit-Anunchai@postgrads.unisa.edu.au
Jonathan.Billington@unisa.edu.au
[2] TextMe Co., Ltd., 20 North Sathorn Rd, Bangkok 10500, Thailand
tul@textme.co.th

Abstract. A new protocol designed for real-time applications, the Datagram Congestion Control Protocol (DCCP), is specified informally in a final Internet Draft that has been approved as an RFC (Request For Comment). This paper analyses DCCP's connection management procedures modelled using Coloured Petri Nets (CPNs). The protocol has been modelled at a sufficient level of detail to obtain interesting results including pinpointing areas where the specification is incomplete. Our analysis discovers scenarios where the client and server repeatedly and needlessly exchange packets. This creates a lot of unnecessary traffic, inducing more congestion in the Internet. We suggest a modification to the protocol that we believe solves this problem.

Keywords: DCCP, Internet Protocols, Coloured Petri Nets, State space methods.

1 Introduction

Streaming media applications and online games are becoming increasingly popular on the Internet. Because these applications are delay sensitive, they use the User Datagram Protocol (UDP) rather than the Transmission Control Protocol (TCP). The users then implement their own congestion control mechanisms on top of UDP or may not implement any control mechanism at all. The growth of these applications therefore poses a serious threat to the Internet. To tackle this problem, an Internet Engineering Task Force (IETF) working group is developing a new transport protocol, called the Datagram Congestion Control Protocol (DCCP) [5–8]. The purpose of DCCP is to support various congestion control mechanisms that suit different applications. It therefore could replace TCP/UDP for delay sensitive applications and become the dominant transport protocol in the Internet. Hence we consider that it is important to verify DCCP as soon as possible, to remove errors and ambiguities and ensure its specification is complete before implementation.

In this paper, Coloured Petri Nets (CPNs) [4] are used to model and analyse DCCP's connection management procedures. We chose CPNs because they are

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 143–158, 2005.

used widely to model and analyse concurrent and complex systems [1,4] including transport protocols like TCP [2,3]. We have previously applied our methodology [2] to earlier versions of DCCP. We demonstrated [11] that a deadlock occurs in DCCP version 5 [6] during DCCP connection setup. Further work [9] upgraded the model to version 6 [7] and also discovered undesired terminal states. These models [11,9] were incomplete, in that they did not include DCCP's synchronisation procedures, which are used in conjunction with connection management.

As far as we are aware, this paper describes the first formal specification of DCCP's connection establishment, close down and synchronisation procedures for version 11 [8] of the specification. Further, using a set of initial configurations, we incrementally analyse the connection management procedures including the synchronization mechanism. Although no deadlock or livelock is found, we discover some chatter in the protocol where both ends repeatedly exchange packets, creating a lot of unnecessary traffic. We canvass a possible solution to this problem. A further contribution of this paper is the identification of areas where the specification is incomplete.

This paper is organised as follows. To make the paper self-contained, section 2 summarises DCCP's connection management procedures. The CPN model of DCCP is illustrated in section 3, which starts with a statement of scope and modelling assumptions, and closes with a discussion of areas of incompleteness in the specification. Section 4 presents our analysis results and section 5 summarises our work.

2 Datagram Congestion Control Protocol

DCCP [8] is a connection oriented protocol designed to overcome the problem of uncontrolled UDP traffic. The connection management procedures have some similarities with TCP, with some states being given the same names. However, DCCP's procedures are substantially different from those of TCP. For example, connection establishment uses a 4-way handshake (rather than 3), there is no notion of simultaneously opening a connection, connection release is simpler, as it does not aim to guarantee delivery of data in the pipeline, and the use of sequence numbers is quite different. There is also a procedure which allows a server to request that the client closes the connection and waits for 2 Maximum packet lifetimes (MPL) to ensure all old packets are removed, before a new instance of the connection can be established, rather than the server having to wait for this period. Further, DCCP includes procedures for resynchronizing sequence numbers. This section summarises the key features of DCCP connection management that we wish to model and analyse.

2.1 DCCP Packet Format

Like TCP, DCCP packets comprise a sequence of 32 bit words as shown in Fig. 1. The DCCP header contains 16 bit source and destination port numbers, an 8 bit data offset, a 16 bit checksum and sequence and acknowledgement numbers in a very similar way to TCP. However, there are significant differences. DCCP

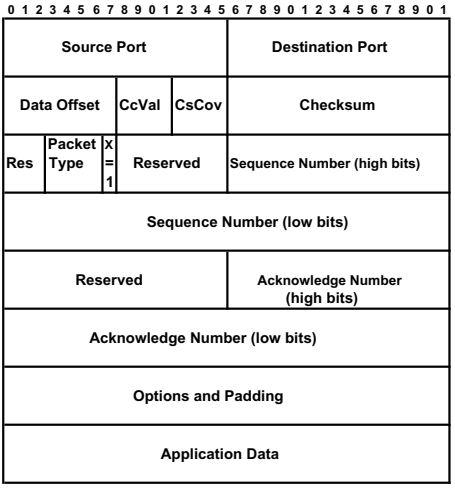

Fig. 1. DCCP Packet Format

defines 10 packets that are encoded using a 4 bit Packet Type field, rather than the control bits used in TCP (for SYN, FIN, RST, ACK). The packets are: Request, Response, Data, DataAck, Ack, CloseReq, Close, Reset, Sync and SyncAck. Sequence (and acknowledgement) numbers are 48 bits long (instead of 32 bits) and number packets rather than octets. The sequence number of a DCCP-Data, DCCP-Ack or DCCP-DataAck packet may be reduced to 24 bits by setting the X field to 0. CCVal, a 4 bit field, contains a value that is used by the chosen congestion control mechanism [8]. Checksum Coverage (CsCov), also a 4 bit field, specifies how much of the packet is protected by the 16 bit Checksum field. Finally, the Options field can contain information such as Cookies and time stamps but also allows DCCP applications to negotiate various features such as the Congestion Control Identifier (CCID) and the size (width) of the Sequence Number validity window [8].

2.2 Connection Management Procedures

The state diagram, shown in Fig. 2, illustrates the connection management procedures of DCCP. It comprises nine states rather than TCP's eleven states. The typical connection establishment and close down procedures are shown in Fig. 3. Like TCP, a connection is initiated by a client issuing an "active open" command. We assume that the application at the server has issued a "passive open" command. After receiving the "active open", the client sends a DCCP-Request packet to specify the client and server ports and to initialize sequence numbers. On receiving the DCCP-Request packet, the server replies with a DCCP-Response packet indicating that it is willing to communicate with the client. The response includes the server's initial sequence number and any features and options that the server agrees to. It also directly acknowledges re-

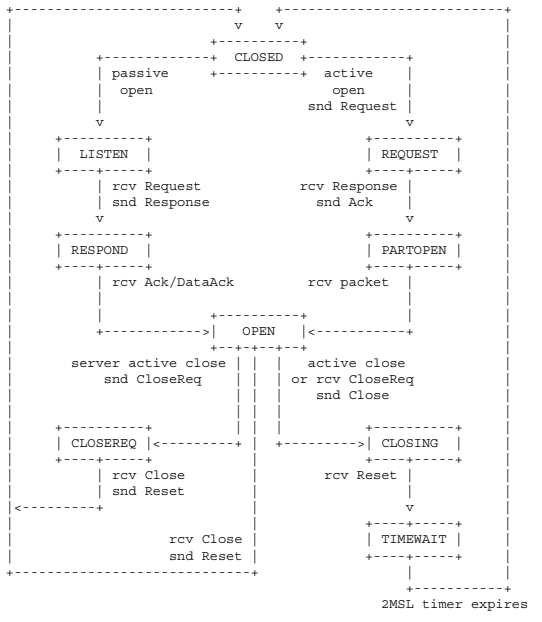

Fig. 2. DCCP State Diagram [8]

ceiving the DCCP-Request. Note that acknowledgements are not cumulative. The client sends an DCCP-Ack or DCCP-DataAck packet to acknowledge the DCCP-Response packet and enters PARTOPEN (this is a new state introduced in version 6 of the protocol). On receiving an acknowledgement from the client, the server enters the OPEN state and is ready for data transfer. At the client, after receiving one of a DCCP-Data, DCCP-DataAck, DCCP-Ack or DCCP-SyncAck packet, the client enters OPEN indicating that the connection is established. During data transfer, the server and client may exchange DCCP-Data, DCCP-Ack and DCCP-DataAck packets (for piggybacked acknowledgements).

Fig. 3 (b) shows the typical close down procedure. The application at the server issues a "server active close" command. The server sends a DCCP-CloseReq packet and enters the CLOSEREQ state. When the client receives a DCCP-CloseReq packet, it must generate a DCCP-Close packet in response. After the server receives a DCCP-Close packet, it must respond with a DCCP-Reset packet and enter the CLOSED state. When the client receives the DCCP-Reset packet, it holds the TIMEWAIT state for 2 MPL[1] before entering the CLOSED state.

Alternatively, either end will send a DCCP-Close packet to terminate the connection when receiving an "active close" command from the application. The end that sends the DCCP-Close packet will hold the TIMEWAIT state

[1] Maximum packet lifetime time (MPL) = Maximum Segment Lifetime (MSL) in TCP.

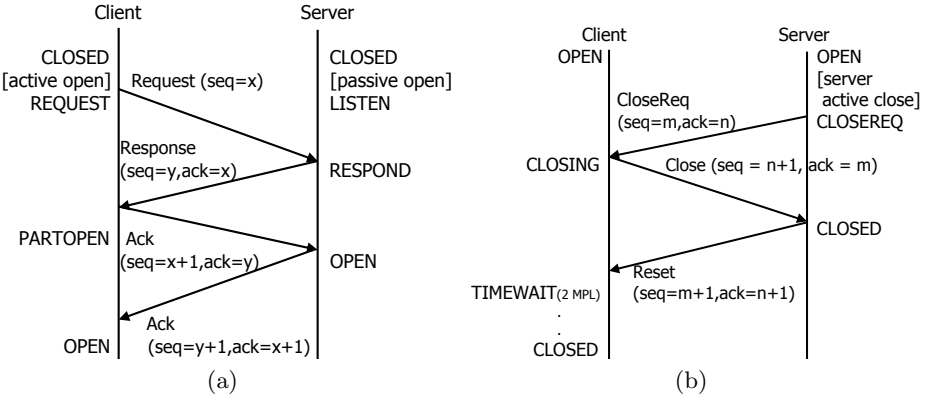

Fig. 3. Typical Connection Establishment and Release Scenarios

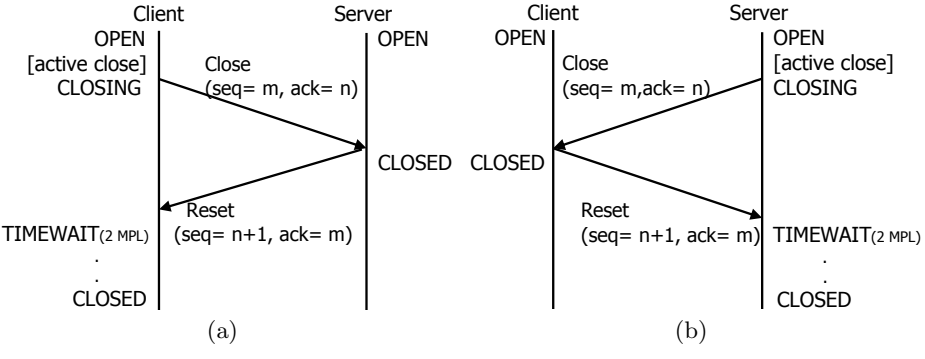

Fig. 4. Alternative Close Down Procedures

as shown in Fig. 4. Beside these three closing procedures, there are another 2 possible scenarios concerned with simultaneous closing. The first procedure is invoked when both users issue an "active close". The second occurs when the client user issues an "active close" and the application at the server issues the "server active close" command.

2.3 Retransmission and Back-Off Timers

Besides the timer (2MPL) in the TIMEWAIT state, DCCP defines two further timers: Retransmission and Back-off. When the sending client does not receive an answer, the timeout period that it waits before retransmitting a packet is determined by the Retransmission Timer (typically 2RTT[2]). After retransmitting for a period (typically 4MPL), it sends a DCCP-Reset and enters the CLOSED state. This timeout period is determined by the Back-off Timer. Generally, if the

[2] RTT = Round Trip Time.

Table 1. Validity Condition for Sequence and Acknowledgement Numbers

Packet Type	Sequence Number Check	Acknowledgement Number Check
Request	SWL $\leq$ seqno $\leq$ SWH	N/A
Response	SWL $\leq$ seqno $\leq$ SWH	AWL $\leq$ ackno $\leq$ AWH
Data	SWL $\leq$ seqno $\leq$ SWH	N/A
Ack	SWL $\leq$ seqno $\leq$ SWH	AWL $\leq$ ackno $\leq$ AWH
DataAck	SWL $\leq$ seqno $\leq$ SWH	AWL $\leq$ ackno $\leq$ AWH
CloseReq	GSR $<$ seqno $\leq$ SWH	GAR $\leq$ ackno $\leq$ AWH
Close	GSR $<$ seqno $\leq$ SWH	GAR $\leq$ ackno $\leq$ AWH
Reset	GSR $<$ seqno $\leq$ SWH	GAR $\leq$ ackno $\leq$ AWH
Sync	SWL $\leq$ seqno	AWL $\leq$ ackno $\leq$ AWH
SyncAck	SWL $\leq$ seqno	AWL $\leq$ ackno $\leq$ AWH

server does not receive a timely response (typically 4MPL), it sends a DCCP-Reset and enters CLOSED. This timeout period is also governed by the Back-off Timer. However when in CLOSEREQ if no response is received within 2 RTT, the server retransmits DCCP-CloseReq. Retransmissions typically occur for 4 MPL but if no response is received, a DCCP-Reset is sent and the server enters the CLOSED state. The sequence number of every retransmitted packet is always increased by one.

2.4 Variables and Sequence Validity

For each connection, DCCP entities maintain a set of state variables. Among those, the important variables are Greatest Sequence Number Sent (GSS), Greatest Sequence Number Received (GSR), Greatest Acknowledgement Number Received (GAR), Initial Sequence Number Sent and Received (ISS and ISR), Valid Sequence Number window width (W) and Acknowledgement Number validity window width (AW). Based on the state variables, the valid sequence and acknowledgement number intervals are defined by Sequence Number Window Low and High [SWL,SWH], and Acknowledgement Number Window Low and High [AWL,AWH] according to the equations of pages 40–41 of the DCCP Definition [8]. Additionally the SWL and AWL are initially not less than the initial sequence number received and sent respectively.

Generally, received DCCP packets that have sequence and acknowledgement numbers inside these windows are valid, called "sequence-valid". Table 1 shows the window ranges for each packet type. The DCCP-CloseReq, DCCP-Close and DCCP-Reset are valid only when seqno $>$ GSR and ackno $\geq$ GAR.

However, there are some exceptions to Table 1, depending on state. According to the pseudo code (see [8] pages 54–58), no sequence validity check is performed in the CLOSED, LISTEN and TIMEWAIT states. In the REQUEST state, only the acknowledgement numbers of the DCCP-Response and DCCP-Reset packets are validated. Other packet types received are responded to with a DCCP-Reset. The acknowledgement number of a DCCP-Reset received in the REQUEST state is validated using [AWL,AWH] instead of [GAR,AWH].

2.5 DCCP-Reset Packets

An entity in the CLOSED, LISTEN or TIMEWAIT state ignores a DCCP-Reset packet while replying to any other unexpected packet types with DCCP-Reset. In other states on receiving a sequence-valid DCCP-Reset packet, the entity goes to TIMEWAIT for 2MPL and then enters the CLOSED state. If the DCCP-Reset packet received is sequence-invalid, the entity responds with a DCCP-Sync. However a sequence-invalid DCCP-Response or DCCP-Reset received in the REQUEST state will be responded to with a DCCP-Reset instead of a DCCP-Sync. When the client is in the REQUEST state, it has not received an initial sequence number (no GSR). In this case the acknowledgement number of the DCCP-Reset is set to zero.

2.6 Resynchronizing Sequence Numbers

Malicious attack or a burst of noise may result in state variables and sequence and acknowledgement number windows being unsynchronized. The DCCP-Sync and DCCP-SyncAck packets are used to update GSR and to resynchronize both ends. When receiving a sequence-invalid packet, an end must reply with a DCCP-Sync packet. It does not update GSR because the sequence number received could be wrong. However the acknowledgement number in the DCCP-Sync packet is set equal to this invalid received sequence number. After receiving a sequence-valid DCCP-Sync, the end must update its GSR variable and reply with a DCCP-SyncAck. It does not update GAR. After receiving a sequence-valid DCCP-SyncAck, an end updates GSR and GAR. An end ignores sequence-invalid DCCP-Sync and DCCP-SyncAck packets, except in the CLOSED, TIMEWAIT, LISTEN and REQUEST states where a DCCP-Reset is sent in response.

3 Modelling DCCP's Connection Management Procedures

3.1 Modelling Scope and Assumptions

Our model comprises all the state transitions of Fig. 2, including the following details from the DCCP Definition [8]: the pseudo code of section 8.5 (pages 54–58); the narrative description of DCCP's event processing in section 8 (pages 48–54); and packet validation in section 7 (pages 38–44). We also make the following assumptions regarding DCCP connection management when creating our CPN model.

1. We only consider at single connection instance while ignoring the procedures for data transfer, congestion control and other feature options. A DCCP packet is modelled by its packet type, sequence number and acknowledgement number. Other fields in the DCCP header are omitted because they do not affect the operation of the connection management procedure.

2. Sequence numbers are assumed not to wrap.

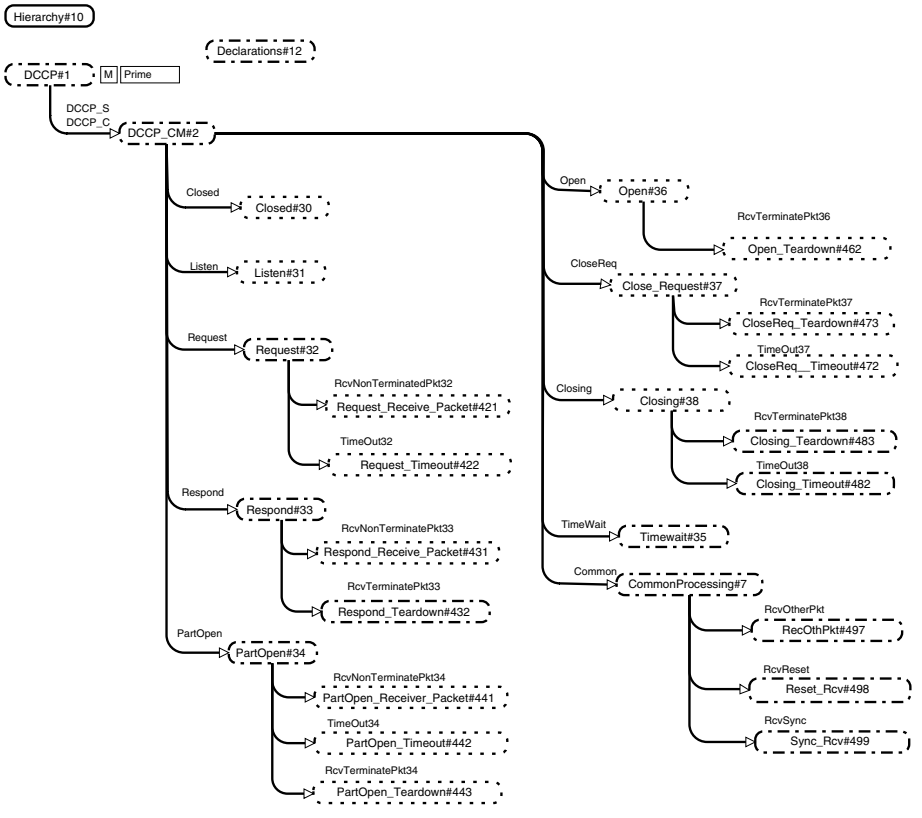

Fig. 5. Hierarchy Page

3. We do not consider misbehaviour or malicious attack.

4. Reordered or lossy channels may mask out possible deadlock, such as unspecified receptions. Thus we incrementally study [2] the CPN model with the following channel characteristics: FIFO without loss, reordered without loss, FIFO with loss, and reordered with loss. However due to space limitations, we only discuss the case when the communication channels can delay and reorder packets without loss.

5. We set the window size to 100 packets because it is specified as the initial default value in the specification (page 31 of DCCP [8]).

6. Without loss of generality, we only use DCCP-Ack and not DCCP-DataAck in order to reduce the size of the state space.

3.2 Structure

The structure of our DCCP CPN model has been influenced by our earlier work [3,11]. It is structured into four hierarchical levels shown in Fig. 5, and comprises 6 places, 27 substitution transitions, 63 executable transitions and 9 func-

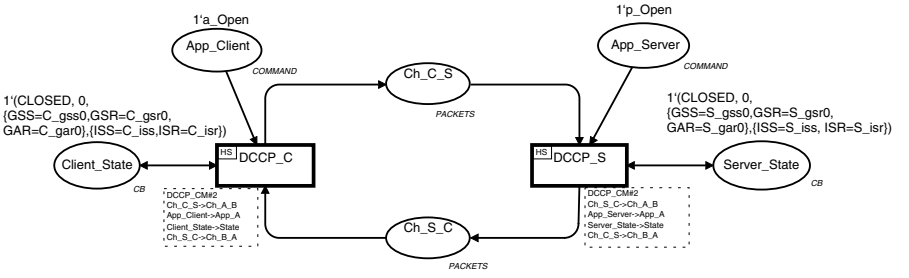

Fig. 6. The DCCP Overview Page

```
1 color PacketType1 = with Request | Data;
2 color PacketType2 = with Sync | SyncAck | Response | Ack | DataAck
                                | CloseReq | Close | Rst;
3 var p_type1:PacketType1;
4 var p_type2:PacketType2;
5 color SN = IntInf with ZERO..MaxSeqNo;            var sn:SN;
6 color SN_AN = record SEQ:SN*ACK:SN;               var sn_an:SN_AN;
7 color PacketType1xSN = product PacketType1*SN;
8 color PacketType2xSN_AN = product PacketType2*SN_AN;
9 color PACKETS = union PKT1:PacketType1xSN+PKT2:PacketType2xSN_AN;
```

Fig. 7. Definition of DCCP PACKETS

tions. The first level is named DCCP. This level calls a page named DCCP_CM (DCCP connection management) twice (for the client and the server). This allows one DCCP entity to be defined and instantiated as either a client or server, greatly simplifying the specification and its maintenance. This has proven to be very beneficial due to there being 6 revisions since we first modelled DCCP [11]. The third level has ten pages, describing the procedures that are followed in each DCCP state. Processing common to several states is specified in the Common Processing page. For convenience of editing and maintaining the model, we group the transitions that have common functions into the fourth level pages. Significant effort has gone into validating this model against the DCCP definition [8] by using manual inspection and interactive simulation [2].

3.3 DCCP Overview

The top level, corresponding to DCCP#1 in the hierarchy page, is the DCCP Overview Page shown in Fig. 6. It is a CPN diagram comprising 6 *places* (represented by ellipses), two *substitution transitions* (represented by rectangles with an HS tag) and *arcs* which connect places to transitions and vice versa. The client is on the left and the server on the right and they communicate via two channels, shown in the middle of Fig. 6. We model unidirectional and re-ordering channels from the client to the server and vice versa by places named Ch_C_S

```
1 color STATE = with CLOSED | LISTEN | REQUEST | RESPOND |
          PARTOPEN | S_OPEN | C_OPEN | CLOSEREQ | CLOSING | TIMEWAIT;
2 color RCNT = int;   var rcnt:RCNT; (*Retransmit Counter *)
3 color GS = record GSS:SN*GSR:SN*GAR:SN;       var g:GS;
4 color ISN = record ISS:SN*ISR:SN;             var isn:ISN;
5 color CB = product STATE*RCNT*GS*ISN;
6 color COMMAND = with p_Open | a_Open | server_a_Close | a_Close;
```

Fig. 8. DCCP's Control Block and User Commands

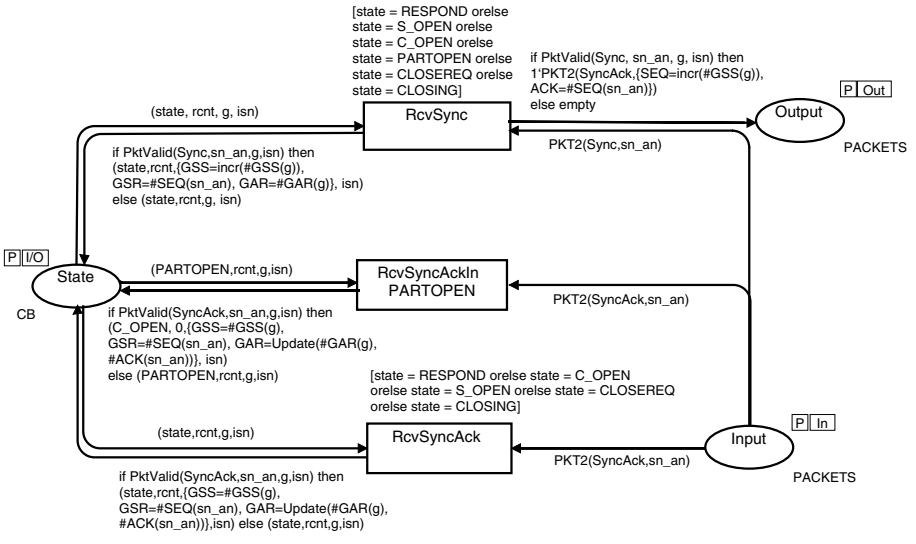

Fig. 9. The Sync_Rcv Page

and Ch_S_C which are typed by PACKETS. Fig. 7 declares PACKETS (line 9) as a union of packets with and without acknowledgements.

Places, Client_State and Server_State, typed by CB (Control Block), store DCCP state information. Fig. 8 defines CB (line 5) as a product comprising STATE, RCNT (Retransmit Counter), GS (Greatest Sequence Number) and ISN (Initial Sequence Number). Places named App_Client and App_Server, typed by COMMAND, model DCCP user commands. Fig. 8 also defines COMMAND (line 6). Tokens associated with these places shown on the top of ellipses, for example "a_Open", are called initial markings. They represent the initial state of the system.

3.4 Second, Third and Fourth Level Pages

The substitution transitions DCCP_C and DCCP_S in Fig. 6 are linked to the second level page named DCCP_CM (as shown in Fig. 5). DCCP_CM is organized into a further ten substitution transitions linked to the third level pages,

```
fun incr(sn:SN) = if (sn = MaxSeqNo)  then ZERO  else IntInf.+(sn, ONE);
fun Update(new:SN,old:SN) = if (IntInf.>(new,old)) then new else old;
val W  = IntInf.fromInt(100); val AW = IntInf.fromInt(100);
fun SeqValid(p_type2:PacketType2, s2:SN_AN, g:GS, isn:ISN) =
let                                (* Sequence Number Validity *)
val SWL=IntInf.max(IntInf.-(IntInf.+(#GSR(g),ONE),
                     IntInf.div(W,IntInf.fromInt(4))),#ISR(isn));
val SWH=IntInf.+(IntInf.+(#GSR(g),ONE),
                    RealToIntInf 4((IntInfToReal 4 W)*3.0/4.0+0.5));
in  case p_type2 of Response => IntInf.>=(#SEQ(s2),SWL)
                                andalso IntInf.<=(#SEQ(s2),SWH)
| Ack =>      IntInf.>=(#SEQ(s2),SWL) andalso IntInf.<=(#SEQ(s2),SWH)
| DataAck =>  IntInf.>=(#SEQ(s2),SWL) andalso IntInf.<=(#SEQ(s2),SWH)
| CloseReq => IntInf.>(#SEQ(s2),#GSR(g)) andalso IntInf.<=(#SEQ(s2),SWH)
| Close =>    IntInf.>(#SEQ(s2),#GSR(g)) andalso IntInf.<=(#SEQ(s2),SWH)
| Rst =>      IntInf.>(#SEQ(s2),#GSR(g)) andalso IntInf.<=(#SEQ(s2),SWH)
| Sync =>     IntInf.>=(#SEQ(s2),SWL)
| SyncAck =>  IntInf.>=(#SEQ(s2),SWL)
| _ => false
 end;
fun AckValid(p_type2:PacketType2, s2:SN_AN, g:GS, isn:ISN) =
 let                               (* Acknowledgement Number Validity*)
  val AWL = IntInf.max(IntInf.-(IntInf.+(#GSS(g),ONE),AW),#ISS(isn));
  val AWH = #GSS(g);
in  case p_type2 of Response =>   IntInf.>=(#ACK(s2),AWL)
                                    andalso IntInf.<=(#ACK(s2),AWH)
| Ack =>     IntInf.>=(#ACK(s2),AWL) andalso IntInf.<=(#ACK(s2),AWH)
| DataAck => IntInf.>=(#ACK(s2),AWL) andalso IntInf.<=(#ACK(s2),AWH)
| CloseReq =>IntInf.>=(#ACK(s2),#GAR(g)) andalso IntInf.<=(#ACK(s2),AWH)
| Close =>   IntInf.>=(#ACK(s2),#GAR(g)) andalso IntInf.<=(#ACK(s2),AWH)
| Rst =>     IntInf.>=(#ACK(s2),#GAR(g)) andalso IntInf.<=(#ACK(s2),AWH)
| Sync =>    IntInf.>=(#ACK(s2),AWL) andalso IntInf.<=(#ACK(s2),AWH)
| SyncAck => IntInf.>=(#ACK(s2),AWL) andalso IntInf.<=(#ACK(s2),AWH)
| _ => false
 end;
fun PktValid(p_type2:PacketType2, s2:SN_AN, g:GS, isn:ISN) =
 SeqValid(p_type2:PacketType2,s2:SN_AN,g:GS,isn:ISN)
 andalso AckValid(p_type2:PacketType2,s2:SN_AN, g:GS, isn:ISN);
```

Fig. 10. Functions used in the Sync Rcv Page

representing the processing required in each DCCP state. We group the transitions that have common functions into the fourth level pages. In particular, we model the behaviour of DCCP in the RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING states when receiving DCCP-Reset and DCCP-Sync packets into pages named Reset_Rcv and Sync_Rcv pages under the Common Processing page in the third level. The RcvOthPkt page models DCCP's behaviour in the OPEN, CLOSEREQ and CLOSING states when receiving pack-

ets other than DCCP-CloseReq, DCCP-Closing, DCCP-Reset, DCCP-Sync and DCCP-SyncAck. Space limits prevent us from including all pages, but a representative example is given in Fig. 9. The figure shows the level of detail required to capture the procedures to be followed by both the client and the server when receiving the DCCP-Sync and DCCP SyncAck packets. Fig. 10 shows functions incr(), Update() and PktValid() used in the Sync_Rcv page.

3.5 Incompleteness in the Specification

User commands appear in the state diagram of Fig. 2 but the specification [8] does not provide any detail. As stated in version 5 [6], the application may try to close during connection establishment. Thus an "active close" command could occur in the REQUEST, RESPOND, PARTOPEN and OPEN states. Similarly, at the server, a "serve active close" command could also occur in the RESPOND and OPEN states. We assume this to be the case in our model, but do not analyse it in this paper.

When the server enters the RESPOND state, it has no information about GAR which is needed to validate the acknowledgement number of DCCP-CloseReq, DCCP-Close and DCCP-Reset. We believe that the specification does not currently cater for the situation when the server receives one of these packets in the RESPOND state. This may happen when the client's user issues an "active close" command while it is in the REQUEST state. The solution to this problem needs further investigation and we do not analyse these scenarios in this paper.

4 Analysis of DCCP CPN Model

4.1 Initial Configuration

Using an incremental approach [2] we analyse different connection management scenarios by choosing a number of different initial markings. This is to gain confidence in the model and to provide insight into DCCP's behaviour. In this paper we limit the maximum number of retransmissions to one to make the generation of the state space tractable. We analyse the DCCP model using Design/CPN 4.0.5 on a Pentium-IV 2 GHz computer with 1GB RAM. Initial markings of each scenario are shown in Table 2.

Case I is for connection establishment. The client and server are both CLOSED with ISS set to five. The client issues an "active Open" command while the server issues a "passive Open" command. There are five scenarios of connection termination when both ends are in the OPEN state. Cases II, III and IV model the case when only one end issues a close command. Cases V and VI represent the simultaneous close scenarios when both ends issue close commands at the same time. Each end has the initial values of GSS, GSR and GAR shown in Table 2, and the channels are empty.

Table 2. Initial Configurations

Case	App_Client	App_Server	Client_State	Server_State
I	1‘a_Open	1‘p_Open	CLOSED	CLOSED
			GSS=0,GSR=0,GAR=0	GSS=0,GSR=0,GAR=0
			ISS=5,ISR=0	ISS=5,ISR=0
II	1‘a_Close		OPEN	OPEN
			GSS=200,GSR=200,GAR=200	GSS=200,GSR=200,GAR=200
III		1‘a_Close	OPEN	OPEN
			GSS=200,GSR=200,GAR=200	GSS=200,GSR=200,GAR=200
IV		1‘Server	OPEN	SOPEN
		a_Close	GSS=200,GSR=200,GAR=200	GSS=200,GSR=200,GAR=200
V	1‘a_Close	1‘a_Close	OPEN	OPEN
			GSS=200,GSR=200,GAR=200	GSS=200,GSR=200,GAR=200
VI	1‘a_Close	1‘Server	OPEN	OPEN
		a_Close	GSS=200,GSR=200,GAR=200	GSS=200,GSR=200,GAR=200

4.2 State Space Results

Table 3 summarizes the state space statistics. The last column shows the number of terminal states which have the same pair of states but different value of GSS, GSR and GAR. In all cases nothing is left in the channels.

In case I, connection establishment, there are three different types of terminal states. The first type is desired when both ends are in OPEN. The second type is when both ends are CLOSED. This can occur when the request or response is sufficiently delayed so that the Back-off timer expires, closing the connection. The third type is when the client is CLOSED, but the server is in the LISTEN state. This situation can happen when the server is initially CLOSED and thus rejects the connection request. After that the server recovers and moves to the LISTEN state. Although we are unable to obtain the full state space for case VI because of state explosion, we can obtain partial state spaces. Case VI a) is when there is no retransmission. Case VI b) is when only one DCCP-Close is retransmitted. Case VI c) is when only one DCCP-CloseReq is retransmitted. Cases II, III, IV, V, VI a), VI b) and VI c) have only one type of terminal state when both ends are in CLOSED.

The Strongly Connected Component (SCC) graphs of all cases (except case VI) were generated. The size of each SCC graph is the same as the size of the state space. This indicates that there are no cycles and hence no livelocks in the state spaces.

4.3 DCCP Chatter During Connection Establishment

Further analysis of case I shows that the state space size grows almost linearly with ISS, as illustrated in Table 4. We have investigated how ISS affects the state space size and found an interesting result. Fig. 11 shows a trace illustrating chatter for the case when ISS=2. The values in brackets, for instance (7,2,3), are (GSS,GSR,GAR). The server, in the CLOSED state, repeatedly sends a

Table 3. State Space Results

Case	Nodes	Arcs	Time (sec)	Dead Markings		
				Client_State	Server_State	Number
I	171,040	457,535	1,067	OPEN	OPEN	67
				CLOSED	CLOSED	1,153
				CLOSED	LISTEN	4
II	73	119	0	CLOSED	CLOSED	8
III	73	119	0	CLOSED	CLOSED	8
IV	30,787	76,796	62	CLOSED	CLOSED	645
V	3,281	8,998	3	CLOSED	CLOSED	64
VI	>545,703	>1,475,936	>152,200	CLOSED	CLOSED	>702
VI a)	437	828	0	CLOSED	CLOSED	33
VI b)	3,324	8,381	3	CLOSED	CLOSED	89
VI c)	33,644	85,926	74	CLOSED	CLOSED	642

Table 4. Growth of the State Space as a Function of ISS

ISS	Nodes	Arcs	Time (sec)	Dead Markings		
				Client_State	Server_State	Number
1	86,058	225,485	325	OPEN	OPEN	67
				CLOSED	CLOSED	733
				CLOSED	LISTEN	4
2	104,464	275,540	457	OPEN	OPEN	67
				CLOSED	CLOSED	823
				CLOSED	LISTEN	4
3	124,763	330,900	596	OPEN	OPEN	67
				CLOSED	CLOSED	923
				CLOSED	LISTEN	4
4	146,955	391,565	785	OPEN	OPEN	67
				CLOSED	CLOSED	1,022
				CLOSED	LISTEN	4
5	171,040	457,535	1,067	OPEN	OPEN	67
				CLOSED	CLOSED	1,153
				CLOSED	LISTEN	4

sequence-invalid DCCP-Reset packet while the client in PARTOPEN repeatedly responds with DCCP-Sync. The sequence and acknowledgement numbers in both packets increase over time until the sequence number of the DCCP-Reset received is greater than the client's GSR and becomes sequence-valid according to Table 1. Fig.11 shows the sequence number of the DCCP-Reset generated increases from zero to three while the client's GSR is equal to two. When receiving the sequence-valid DCCP-Reset (with seq=3), the client enters the TIMEWAIT state and then CLOSED after 2 MPL. A similar situation happens when the server enters the LISTEN state after sending the DCCP-Reset with sequence number zero. This behaviour creates unnecessary traffic, adversely affecting congestion in the Internet. It will be particularly severe if the initial sequence number is even moderately large, which will often be the case.

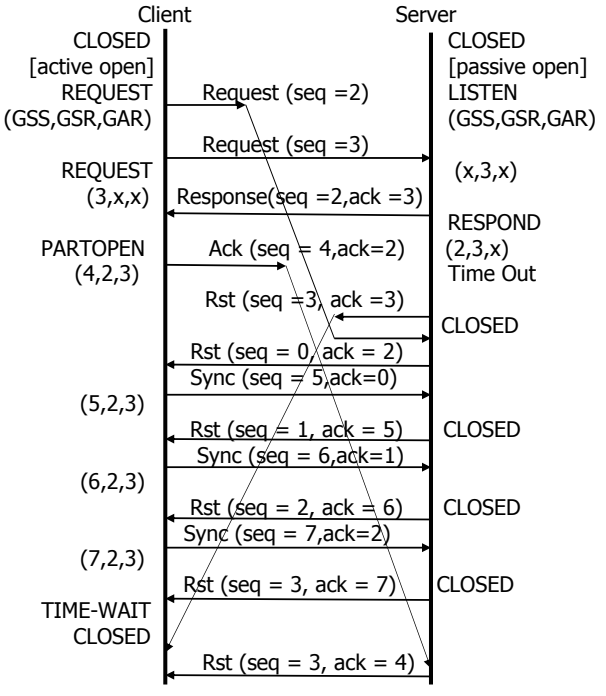

Fig. 11. Repeatedly Exchanged Messages for Case I with ISS=2

This problem is caused by the invalid DCCP-Reset packet having sequence number zero. Because there are no sequence number variables in the CLOSED or LISTEN state, according to the specification [8] section 8.3.1, the sequence number of the DCCP-Reset packet generated in the CLOSED and LISTEN states is the received acknowledgement number plus one. If there is no received acknowledgement number because the received packet type is DCCP-Request (or DCCP-Data), the sequence number of the DCCP-Reset packet is set to zero and the acknowledgement number is set to the received sequence number.

In versions 5 and 6 of the draft specification[6,7], the DCCP-Reset packet with sequence number zero is specified as a valid packet. However, our previous work [9,11] shows that this valid DCCP-Reset causes deadlocks where the server is in the CLOSED state and the client is in OPEN. Since draft specification version 7, a DCCP-Reset with sequence number zero is no longer considered a valid packet. A solution may be to ignore an incoming packet without an acknowledgement number when received in the CLOSED or LISTEN state. This is because every state (except CLOSED and LISTEN) has a Back-off timer, which will guarantee that the other end will eventually go to the CLOSED state.

5 Conclusion

This paper has illustrated a formal specification and has provided an initial but detailed analysis of DCCP's connection management procedures. Signifi-

cant effort has been spent on ensuring that the CPN specification accurately captures the pseudo code and narrative description in the final Internet Draft version 11 [8]. This has revealed areas in the specification which we believe to be incomplete as discussed in section 3.5. Our analysis has discovered scenarios where the client keeps sending DCCP-Sync packets in response to the server sending sequence-invalid DCCP-Reset packets. This may have an adverse effect on congestion in the Internet, if the initial sequence number chosen is even moderately large. Future work will involve modifying the procedures to eliminate this problem and verifying that the revised model works correctly. We need to analyse our model when an application closes during connection establishment as was discussed in section 3.5. We would also like to extend our work to include Option/Feature negotiation.

References

1. Billington, J., Diaz, M. and Rozenberg, G., editors (1999), *Application of Petri Nets to Communication Networks*, Advances in Petri Nets, LNCS Vol 1605, Springer-Verlag.
2. Billington, J., Gallasch, G.,E. and Han, B. (2004), "A Coloured Petri Net Approach to Protocol Verification" in *Lectures on Concurrency and Petri Nets*, LNCS Vol 3098, pp. 210-290, Springer-Verlag.
3. Han, B. and Billington, J. (2002), "Validating TCP connection management" in *Workshop on Software Engineering and Formal Methods, Conferences in Research and Practice in Information Technology*, ACS, Vol. 12, pp. 47-55.
4. Jensen, K. (1997), *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Volumes1-3, Monographs in Theoretical Computer Science, Springer-Verlag, Berlin.
5. Kohler, E. and Floyd, S. (2003), "Datagram Congestion Control Protocol (DCCP) overview", http://www.icir.org/kohler/dcp/summary.pdf.
6. Kohler, E. and Floyd, S. (2003), *Datagram Congestion Control Protocol, Internet-Draft version 5.* draft-ietf-dccp-spec-05.ps, October 2003, http://www.icir.org/kohler/dcp/.
7. Kohler, E. and Floyd, S. (2004), *Datagram Congestion Control Protocol, Internet-Draft version 6.* draft-ietf-dccp-spec-06.ps, February 2004, http://www.icir.org/kohler/dcp/.
8. Kohler, E. and Floyd, S. (2005), *Datagram Congestion Control Protocol, Internet-Draft version 11.* draft-ietf-dccp-spec-11.ps, March 2005, http://www.icir.org/kohler/dcp/.
9. Kongprakaiwoot, T. (2004), *Verification of the Datagram Congestion Control Protocol Using Coloured Petri Nets.* Master of Engineering Minor Thesis, Computer Systems Engineering Centre, University of South Australia.
10. University of Aarhus (2004), Design/CPN Online. http://www.daimi.au.dk/designCPN/.
11. Vanit-Anunchai, S. and Billington, J. (2004), "Initial Result of a Formal Analysis of DCCP Connection Management", *Proc. Fourth International Network Conference*, Plymouth, UK, 6-9 July 2004, pp. 63–70.