**Homework 4: A more involved example**

The last homework of the course, is to make an abstract model of the following protocol, so that SPIN can be used to check that the protocol does not reorder or duplicate messages, in the presence of message losses, duplications, or process crashes. The main work is to make the abstract model readable, so that it has a sufficiently small state space that SPIN can check it exhaustively. In class, we will devote a couple of lectures to tricks and things that can be used.

Deadline: is March 17.

The protocol in question is the Five Packet Handshake Protocol, which is used in TCP for transmission of single messages. Descriptions of the protocol appear, e.g., in [?] and [?, pp. 718–729].

The protocol is intended to transmit single messages from a sender to a receiver. Before each message transmission, a pair of initialization messages must be exchanged in order to establish an appropriate sequence number, which will be associated with the message in question. After the transmission of a message, its receipt must be appropriately acknowledged, again by a pair of messages. Thus, five messages are required for the transmission of a single message, hence the name of the protocol.

The protocol is intended to work in the presence of losses, duplications, reorderings, and arbitrary delays in the channel. Additionally, the sender and receiver may crash, and be forced to reinitialize. Under these liberal conditions, the protocol may sometimes lose a message, but never transmit duplicates. It is well-known [?, Thm 22.13] that no protocol can implement a perfect FIFO channel under these assumptions.

Let us give a more detailed description of the protocol. The protocol consists of a Sender, a Receiver, and a communication medium, which can reorder, lose, and duplicate messages. The Sender receives a stream of messages from the environment, and it is the task of the protocol to transmit these in order to the Receiver, which then forwards them to its environment. In order to recover from crashes, both the Sender and the Receiver maintain a set of unique identifiers (UIDs), taken from an infinite set, in stable memory (i.e., this set is not affected by crashes). The set represents the set of UIDs that have previously been used, and shall not be used again.

The transmission of a message consists of exchanging five packets.

1. The **Sender** sends a packet of form needuid\((v)\), in which \(v\) is a fresh UID. This is a request for a UID from the Receiver to be used for the message transmission.

2. The **Receiver** sends a packet of form accept\((u, v)\) where \(v\) is the UID received in the previous needuid message, and \(u\) is a fresh UID to be used for the message transmission.

3. The **Sender** sends the message \(m\) in a packet of form send\((m, u)\), where \(u\) is the UID just received in the accept message.

4. The **Receiver** acknowledges the message by an ack\((u)\) message.
5. The Sender closes the packet exchange by a cleanup($u$) message, which also tells the Receiver to stop using the UID $u$ in any future packet exchange.

In order to recover from packet losses, any packet can be retransmitted if the next expected packet does not arrive within some time. In the description in [?, pp. 718–729], there is a difference in retransmission policy between different packet types: packets of type needuid, accept, and send can be retransmitted an arbitrary number of times, whereas an ack packet is transmitted only on the receipt of a send packet; this is done even if the send packet is for an “outdated” UID. A cleanup packet can be sent in two situations.

- on the receipt of an ack packet,
- on the receipt of an “outdated” accept packet.

When a Sender or a Receiver crashes, they return to their initial state, but keep the record of used UIDs. In a crash, the Sender may lose some messages that were scheduled for transmission to the Receiver.

In the following, we give a naive simplified Promela model of the protocol, which is taken rather directly from the model of [?, pp. 718–729]. For readability, we have here not included some aspects of the model:

- We have not included our modeling of the imperfections in the channels (loss, re-ordering, duplication). These can be modeled in different ways: either by changing the Promela code that sends and/or receives messages, or by adding a demon process which scrambles the contents of Channels.

- In the analyzed model, each message reception and the following sequence of local operations of a process are included within atomic brackets. A standard rule of thumb is to enclose any sequence triggered by a receive, potentially containing a resulting send, in atomic brackets. In the version shown here, we have omitted the atomic brackets for readability.

Some modeling conventions must be made before the protocol description can be turned into a Promela Model:

- The sequence of messages to be transmitted from Sender to Receiver will be the sequence of numbers 0, 1, 2, . . . upto a maximum number MaxMsg.
- The sequence of UIDs used will similarly be chosen as the sequence 0, 1, 2, . . . upto a maximum number
- We use a separate channel for each packet type, e.g., the channel Sendchan to carry packets of form send($m, u$).

Following is the naive Promela model of the protocol.

```c
#define NULL 0 /* Undefined value of lastUID */
#define MaxSeq 200 /* How many messages to check*/
```
#define ChanSize
5
/* channel size */

chan Needuidchan = [ChanSize] of { byte };
chan Acceptchan = [ChanSize] of { byte , byte };
chan Sendchan = [ChanSize] of { byte , byte };
chan Ackchan = [ChanSize] of { byte };
chan Cleanupchan = [ChanSize] of { byte };
active proctype Sender()
{
  byte SaccUID, /* UID used to get new sequence number */
  SmsgUID, /* UID used as sequence number */
  SnextMsg; /* The message to be transmitted */
  byte u,v; /* Used to receive parameters of messages */

Sidle: SnextMsg < MaxSeq -> SnextMsg++ ; /* get next message to send*/
  SaccUID < MaxSeq -> SaccUID++ ; /* get fresh UID*/
Sneeduid: do
  :: Needuidchan! SaccUID /* (re)transmit first packet */
  :: Acceptchan? u, v -> /* on reception of accept message */
    if
    :: v == SaccUID -> /* if correct uid start sending */
      SmsgUID = u ; break
    :: else -> /* otherwise send cleanup */
      Cleanupchan ! u
    fi
  :: Ackchan? u -> /* on a spurious ack */
    Cleanupchan ! u /* reply with cleanup */
  :: goto Scrash /* crash */
  od;
Ssend: do
  :: Sendchan!SnextMsg,SmsgUID /* (re)transmit message */
  :: Ackchan? u -> /* on reception of ack */
    if
    :: (u == SmsgUID) -> /* if correct uid */
      Cleanupchan!u; /* send cleanup and restart */
    goto Sidle
    :: else -> Cleanupchan!u /* otherwise send cleanup */
    fi
  :: Acceptchan? u,v -> /* if spurious accept */
    if
    :: (u != SmsgUID) -> /* if old, send cleanup */
      Cleanupchan!u
    :: else -> skip /* if current, do nothing */
    fi
  :: goto Scrash /* crash */
  od;
Scrash: do /* lose some input msgs */
  :: SnextMsg < MaxSeq -> SnextMsg++
  :: skip -> break
  od;
  goto Sidle
}
active proctype Receiver()
{
    byte RaccUID, /* UID used to ge new sequence number */
    RmsgUID, /* UID used as sequence number */
    RlastUID, /* remembers last sequence number */
    RexpMsg; /* The message to be received */
    byte m,u,v; /* Used to receive parameters of messages */

    Ridle: RmsgUID < MaxSeq -> RmsgUID++ ; /* get fresh sequence number */
    do
        :: Needuidchan? RaccUID -> /* when needuid arrives */
            break /* start sending accept */
        :: Sendchan?m,u -> /* spurious send */
            if /* if old uid arrives */
                :: (u != RlastUID) -> Ackchan!u /* send ack */
                :: else -> skip
            fi
        :: Cleanupchan?u -> skip /* ignore cleanup */
        :: goto Rcrash /* crash */
    od;

    Raccept: do
        :: Acceptchan! RmsgUID , RaccUID /* (re)transmit msg 2 */
        :: Sendchan ? m , u -> /* on reception of send */
            if
                :: (u == RmsgUID) -> /* if correct uid */
                    RlastUID = u; /* remember uid */
                    assert(m >= RexpMsg); /* check ordering */
                    RexpMsg = m+1; break /* update expected Msg */
            :: else -> skip
            fi
        :: Needuidchan? v -> skip /* ignore needuid */
        :: Cleanupchan? u ->
            if
                :: (u == RmsgUID) -> /* on cleanup */
                    RlastUID = NULL; /* clean RlastUID */
                    goto Ridle
                :: else -> skip
            fi
        :: goto Rcrash /* crash */
    od;
Rack: do
  :: Ackchan!RmsgUID /* (re)transmit ack */
  :: Cleanupchan?u -> /* when cleanup arrives */
      if
        :: (u == RlastUID) -> /* if current uid */
          RlastUID = NULL; /* restart */
          goto Ridle
        :: else -> skip /* else skip */
      fi
  :: Sendchan ? m , u -> /* spurious send msg */
      if
        :: (u != RlastUID) -> /* if old uid */
          Ackchan!u /* send ack */
        :: else -> skip
      fi
  :: Needuidchan? v -> skip /* ignore needuid */
  :: goto Rcrash /* crash */
od;
Rcrash: RlastUID=NULL;
        goto Ridle
}
