

Chapter 3

Reliable Channels from Unreliable Channels

3.1 Introduction

This chapter illustrates SESF with a distributed system example. Different kinds of message-passing channel services are described, as well as sliding window protocol systems that offer reliable channels using unreliable channels. This example also illustrates how time-constrained services and systems can be modeled and analyzed.

We start by specifying four types of channels: `fifo`, `lossy`, `LRD` (loss, reordering, duplication), and `LRDml` (`LRD` with maximum message lifetime). A `fifo` channel delivers messages in the order sent. A `lossy` channel is a `fifo` channel except that it can lose messages in transit. A `LRD` channel can lose, reorder, and duplicate messages in transit. A `LRDml` channel is a `LRD` channel with a maximum message lifetime parameter `L`: within `L` seconds of a message being sent, that message, including any duplicate, is no longer in the channel. The progress obligation of a `fifo` channel is that every message sent is eventually delivered. The progress obligation of a `lossy`, `LRD` or `LRDml` channel is that a message repeatedly sent is repeatedly delivered.

We then present a distributed system that offers `fifo` channels using `lossy` channels. The system implements the sliding-window protocol with cyclic, or modulo-`N`, sequence numbers. The minimum `N` needed to achieve correctness is derived as a function of the sliding window sizes.

Finally, we show that imposing a time constraint on the distributed system, specifically, a minimum time between data transmissions, results in a system that offers `fifo` channels using `LRDml` channels. The minimum `N` needed to achieve correctness is derived as a function of the sliding window sizes, the maximum message lifetime `L`, and the minimum time between data transmissions.

The specifications involve sequences of messages and common operations on sequences, e.g., concatenation, prefix, subsequence. We use “`<`” and “`>`” to define sequences, for example, `<>` is the empty sequence and `<a, b, c>` is a three element sequence with `a` being the first and `c` the last. We use “`o`” for concatenation, so `<a> o <b, c>` is equal to `<a, b, c>`. For a finite sequence `x`, `x.size` denotes the number of elements in `x`. For a sequence `x`, we refer to the `j`th entry as `x[j]`, where `j = 0` representing the head. We denote the indices of `x` by `x.domain`; so `x.domain = 0..x.size - 1` for a finite sequence `x`.

3.2 Point-to-point channel services

Below, we specify four kinds of point-to-point channels. A point-to-point channel transfers messages from one location to another. In the following service specifications, `x` identifies the transmitter, `y` identifies the receiver, `Msg` denotes the messages that can be sent and received. For notational convenience, we allow `Msg` to have records. (In practice, `Msg` usually consists of bounded-size byte sequences, and so sending structured data involves data encoding and decoding functions known to both end points).

Fifo channels. We start by specifying `fifo` channels.

```

service-program FifoChannel( x, y, Msg ) {
  init() {
    MsgSeq txh := ⟨ ⟩; // sequence of messages transmitted at x
    MsgSeq rxh := ⟨ ⟩; // sequence of messages received at y
  }

  dnw-event Tx( Msg msg ) {
    ec true
    ac txh := txh ◦ ⟨ msg ⟩
  }

  upw-event Rx( Msg msg ) {
    ec ( rxh ◦ ⟨ msg ⟩ ) prefix-of txh
    ac rxh := rxh ◦ ⟨ msg ⟩
  }

  progress-obligation() { // every message sent is eventually received
    [ forall int i : txh.size ≥ i ⇔ rxh.size ≥ i ]
  }
} // FifoChannel

```

Note that the service specification is not concerned with how the service is implemented. For example, the transmitter and receiver may be connected by an error-free physical link. Or they may be connected by an error-prone physical link that runs a retransmission-with-acknowledgement protocol (e.g., HDLC). Or they may be connected via a network of intermediate store-and-forward nodes that run routing, data forwarding, and end-to-end data transfer protocols (e.g., OSPF, IP, TCP).

Note also that the service specification does explicitly indicate the messages in transit. But this is easily obtained from txh and rxh : the sequence of messages in transit is simply the last $txh.size - rxh.size$ messages in txh .

Lossy channels. A lossy channel service is obtained by modifying fifo channel service in two ways. First, the enabling condition of the receive event changes: “prefix of” becomes “subsequence of”. Second, the progress obligation now states that a message repeatedly sent is repeatedly delivered. We use $nbr(h, m)$ to denote the number of occurrences of element m in sequence h .

```

service-program LossyChannel( x, y, Msg ) {
  init() {
    MsgSeq txh := ⟨ ⟩; // sequence of messages transmitted
    MsgSeq rxh := ⟨ ⟩; // sequence of messages received
  }

  dnw-event Tx( Msg msg ) {
    ec true
    ac append( txh, msg )
  }

  upw-event Rx( Msg msg ) {
    ec ( rxh ◦ seq(msg) ) subsequence-of txh
    ac append( rxh, msg )
  }

  progress-obligation() {
    [forall Msg m: // a message repeatedly sent is repeatedly received
      [forall int i: nbr(txh, m)=i ⇔ nbr(txh, m)>i ]
      ⇒ [forall int i: nbr(rxh, m)=i ⇔ nbr(rxh, m)>i ]
    ]
  }
}

```

```

    ]
  }
} // LossyChannel

```

As in the fifo case, the service specification does explicitly indicate the sequence of messages in transit. Unlike in the fifo case, the sequence is not uniquely determined by txh and rxh because of losses. For example, if $\text{txh} = \langle a, b, c, d \rangle$ and $\text{rxh} = \langle a, b \rangle$, then either $\langle c, d \rangle$, $\langle d \rangle$, $\langle c \rangle$, or $\langle \rangle$ can be in transit. But we can say that a message m is potentially in transit if $\text{rxh} \circ \langle m \rangle$ subsequence-of txh holds.

LRD channels. A LRD channel service is obtained from the lossy channel service by changing the enabling condition of $\text{Rx}(m)$ to “ m in txh ”, thereby allowing any message that has been sent to be received at any time. No other change is needed. Here we can say that a message m is potentially in transit if m in txh holds.

LRDml channels. A LRDml channel is like a LRD channel except that it has a maximum message lifetime parameter, say L (typically orders larger than the average delay). L seconds after a message is sent, that message, including any duplicate, is no longer in the channel and hence cannot be received at a later point in time.

To model a time-constrained service, we need a model of real time. Let a real-valued variable τnow indicate the current time. Introduce an event $\text{Age}(\Delta)$ that increases τnow by an arbitrary positive real value Δ . We group them in a hypothetical system program called `RealTime`:

```

system-program RealTime( ) {
  real  $\tau\text{now} := 0$ ;

  lc-event Age( real  $\Delta$  ) {
    ec  $\Delta > 0$ 
    ac  $\tau\text{now} := \tau\text{now} + \Delta$ 
  }
}

```

A time-constrained system or service can now be modeled by using the value of τnow in event enabling conditions and actions. In $\text{Tx}(\text{msg})$, append $\langle \text{msg}, \tau\text{now} \rangle$ to txh , so that txh records the sequence of messages sent along with their transmit times. Similarly, in $\text{Rx}(\text{msg})$, append $\langle \text{msg}, \tau\text{now} \rangle$ to rxh , so that variable rxh records the sequence of messages received along with their receive times. Finally, in $\text{Rx}(\text{msg})$, change the enabling condition to $((\text{msg}, t) \text{ in } \text{txh})$ and $(\tau\text{now} - t < L)$, so that only messages younger than L seconds are received. This results in the following service program, which exists in the context of the `RealTime` program above.

```

service-program LRDmlChannel( x, y, Msg, L ) { // LRD with max message lifetime L
  init() {
    MsgTimeSeq txh :=  $\langle \rangle$ ; // sequence of (message, time) pairs transmitted
    MsgTimeSeq rxh :=  $\langle \rangle$ ; // sequence of (message, time) pairs received
  }

  dnw-event Tx( Msg msg ) {
    ec true
    ac txh := txh  $\circ \langle (\text{msg}, \tau\text{now}) \rangle$ ;
  }

  upw-event Rx( Msg msg ) {
    ec (msg, t) in txh and  $(\tau\text{now} - t < L)$ 
    ac rxh := rxh  $\circ \langle (\text{msg}, \tau\text{now}) \rangle$ ;
  }

  progress-obligation() {

```

```

[forall Msg m: // a message repeatedly sent is repeatedly received
  [forall int i: nbr(txh, m)=i  $\rightsquigarrow$  nbr(txh, m)>i ]
   $\Rightarrow$  [forall int i: nbr(rxh, m)=i  $\rightsquigarrow$  nbr(rxh, m)>i ]
]
}
} // LRDml channel

```

Here we can say that a message m is potentially in transit if $((m, t) \text{ in } txh)$ and $(\tau_{\text{now}} - t < L)$ holds.

3.3 Offering fifo channels using lossy channels

This section presents a distributed system intended to offer fifo channels using lossy channels. The system employs the sliding-window protocol with cyclic, or modulo- N , sequence numbers. The minimum N needed to achieve correctness is derived as a function of the sliding window sizes.

Our distributed system, shown in Figure 3.1, consists of two systems, an entity at x and an entity at y , that cooperate to offer a fifo channel from x to y , making use of lossy channels between x and y . The system can be duplicated in the reverse direction to offer a fifo channel from y to x .

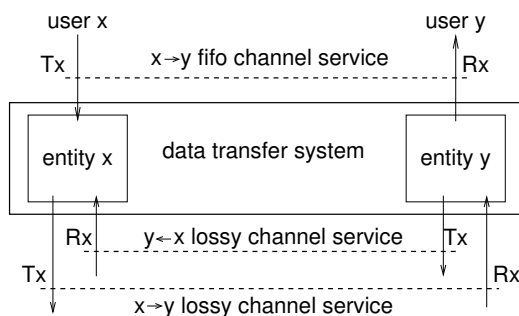


Figure 3.1: Data transfer system offering fifo channel using lossy channels.

User x passes data blocks to entity x to be delivered to user y . Because the channels can lose messages, entity x has to retransmit every data block until it receives an acknowledgement from entity y . So entity x has to buffer all data blocks passed down by user x that have not yet been acked. Because the fifo channel service to be offered is non-blocking, the amount of data that entity x may have to buffer is unbounded (as we shall see, this can be fixed with flow control).

We point that, for throughput reasons, we want entity x to be able to have several data blocks *outstanding*, i.e., sent but not acknowledged. Similarly, entity y should be able to buffer data blocks received out of sequence (due to retransmissions by entity x).

Clearly, the two entities have to agree on some way of identifying data blocks. One obvious way is for entity x to associate successive data blocks provided by user x with increasing sequence numbers, and to include the sequence number when it sends a data block. Entity y can then determine the correct position of any received data block in the stream of data blocks it delivers to user y . Entity y can also send back the sequence number to acknowledge reception of the data block. The only problem is that such a sequence number field would grow without bound as more and more is transferred. This is not desirable because it wastes channel bandwidth and requires more processing resources in the end points.

The typical solution is to use “cyclic”, or modulo- N , sequence numbers in place of the “unbounded” sequence numbers. So when entity x sends data block j , it attaches the cyclic sequence number $j \bmod N$, which we write as $\text{mod}(j, N)$. When entity y receives a cyclic sequence number, it must correctly infer the corresponding unbounded sequence number. Similarly, acknowledgements also can use cyclic sequence numbers. Clearly, this can work properly only if the received sequence numbers stay within appropriate bounds (otherwise a cyclic sequence number may correspond to several unbounded sequence numbers).

The sliding-window protocol achieves this in an elegant way. It maintains two “windows” of sequence numbers, as shown in Figure 3.2. The “send” window at entity x corresponds to the sequence numbers of the

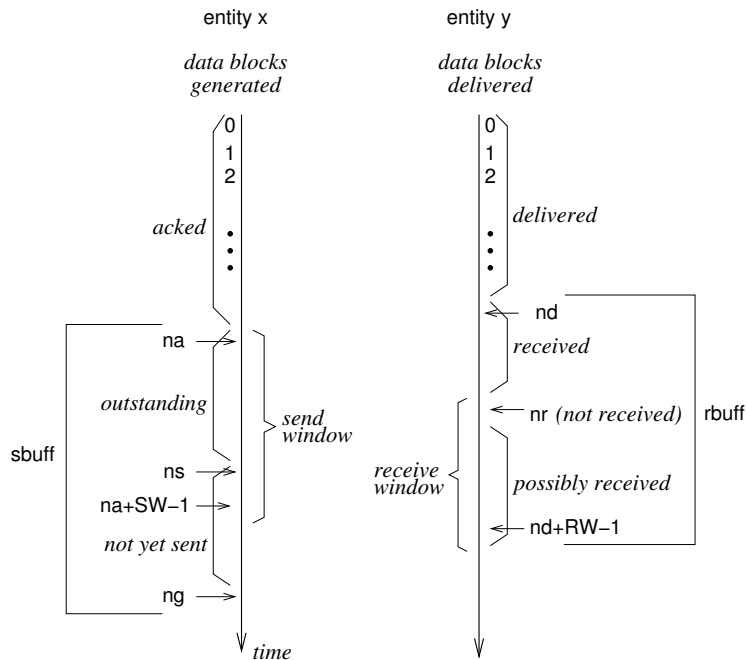


Figure 3.2: Sliding window mechanism.

datablocks that entity x can have outstanding. Its size is SW . The “receive” window at entity y corresponds to the sequence numbers of the datablocks that entity y is prepared to receive out of sequence. Its size is RW .

Entity x maintains the following variables: ng , the number of data blocks passed down by the local user; ns , the number of data blocks sent at least once; na , the number of data blocks acknowledged; and $sbuff$, a buffer storing data blocks na to $ng - 1$ (i.e., those passed down by the local user and not yet acked). We treat $sbuff$ as a sequence of entries indexed 0 through $ng - na - 1$, where $sbuff[i]$ has data block $na + i$.

Data blocks 0 through $na - 1$ have been sent and acked. Data blocks na through $ns - 1$ are outstanding. The sequence numbers na through $na + SW - 1$ constitute the send window. At all times, $na \leq ns \leq ng$ and $ns - na \leq SW$ hold. So at most SW data blocks can be outstanding. Note that $na + SW$ can be higher or lower than ng .

Entity y maintains the following variables: nd , the number of data blocks delivered to the local user; nr , the number of contiguous data blocks received from the remote user; and $rbuff$, a buffer for holding received data blocks nd through $nr + RW - 1$. We treat $rbuff$ as a sequence of entries indexed 0 through $nr - nd + RW - 1$, where entry i either has data block $nd + i$ or is empty.

Data blocks 0 to $nd - 1$ have been delivered in order to the local user. Data blocks nd to $nr - 1$ are ready for delivery to the local user, and are buffered until then. At all times, $nd \leq nr$ holds. Data block nr has not been received. Data blocks in the range $nr + 1$ to $nr + RW - 1$ have been received out of sequence and are being buffered. The sequence numbers nr to $nr + RW - 1$ constitute the receive window.

Each data message sent by entity x consists of a data block j and the cyclic sequence number $\text{mod}(j, N)$. Each ack message sent by entity y consists of the cyclic sequence number $\text{mod}(nr, N)$. As in most sliding window protocols, the acknowledgement message sent in response to a received data block j is *not* the received cyclic sequence number $\text{mod}(j, N)$ but rather $\text{mod}(nr, N)$, i.e., the cyclic sequence number of the data block next expected in sequence. Such acknowledgements are referred to as “cumulative”, because they ack all earlier data blocks (in fact, a better name would be “request for data”).

When entity y receives a data block with cyclic sequence number cj , it looks for an unbounded sequence number j in the receive window that matches (i.e., $cj = \text{mod}(j, N)$); if such a j exists, it treats cj as corresponding to that j . Note that the first unbounded number on or after nr that matches cj is $nr + \text{mod}(cj - nr, N)$, and this number is in the receive window iff $\text{mod}(cj - nr, N) < RW$ holds.

Similarly, when entity x receives a cyclic sequence number c_j , it looks for a matching unbounded sequence number j in the window of outstanding sequence numbers, i.e., j in $na+1, \dots, ns$ such that $\text{mod}(j, N)$ equals c_j ; if such a j exists it treats c_j as corresponding to that j . Again, the first unbounded number on or after na that matches c_j is $na + \text{mod}(c_j - na, N)$, and this number is in the “outstanding” window iff $\text{mod}(c_j - na, N) \leq ns - na$.

```

system-program SWSrc(x, y, N, Data ) { // executed by entity x
  init() {
    int ng := 0; // number of data blocks accepted from local user
    int ns := 0; // number of data blocks sent at least once
    int na := 0; // number of data blocks acknowledged
    Buffer sbuff := ⟨⟩; // buffer holding data blocks na, ⋯, ng - 1
  }

  xc-event Accept(Data d) {
    maps FifoChannel(x,y,Data).Tx(d)
    ec true
    ac append( sbuff, d );
    ng := ng + 1;
  }

  lc-event Send() {
    ec ns < ng and ns < na + SW
    ac LossyChannel(x,y,DMsg).Tx( DMsg(sbuff[ns-na], mod(ns,N)) );
    ns := ns + 1;
  }

  lc-event Resend(int j) {
    ec 0 ≤ j ≤ ns-na-1
    ac LossyChannel(x,y,DMsg).Tx( DMsg(sbuff[j], mod(na+j,N)) );
  }

  xc-event Receive(AMsg msg) {
    maps LossyChannel(y,x,AMsg).Rx( AMsg msg )
    ec true
    ac int tmp := mod( msg.csn-na, N);
    if 1 ≤ tmp ≤ ns-na then { // csn matches na+tmp
      sbuff := sbuff[ tmp .. ng-1 ]; // delete entries 0, ⋯, tmp - 1
      na := na + tmp;
    }
  }

  progress-assumptions { weak fairness for Send and Resend }
} // SWSrc

```

Figure 3.3: Source entity of the sliding window protocol.

The programs of the source and sink entities are shown in Figures 3.3 and 3.4. Note that locally-controlled activity is specified entirely by lc events, without explicitly defining any processes. The type `Data` denotes the data blocks that may be exchanged by the users. The types `DMsg` and `AMsg` denote, respectively, the data messages and the ack messages. `DMsg` is a record type with two fields: `data` of type `Data`, and `csn` (cyclic sequence number) of type `0..N-1`. `AMsgs` is a record type with a field `csn` of type `0..N-1`. For any particular values d and c_j from `Data` and `0..N-1`, respectively, `DMsg(d, cj)` denotes the corresponding data message. Likewise with `AMsg(cj)`.

```

system-program SWSink(y, x, N, Data ) { // executed by entity y
  init() {
    int nd := 0; // number of data blocks delivered to local user
    int nr := 0; // number of data block next expected in sequence
    Buffer rbuff := RW empty entries; // buffer for holding data blocks nd, ..., nd + RW - 1
  }

  lc-event Deliver() {
    ec nd < nr
    ac FifoChannel(x,y,Data).Rx(rbuff[0]);
    rbuff := rbuff[1..nr-nd+RW-1] o <"empty"; // delete entry 0, append empty entry
    nd := nd + 1;
  }

  lc-event Sendack() {
    ec true
    ac LossyChannel(y, x, AMsg).Tx( AMsg(mod(nr,N)) );
  }

  xc-event Receive(DMsg msg) {
    maps LossyChannel(x,y,DMsg).Rx( DMsg msg )
    ec true
    ac int tmp := mod( msg.csn - nr, N);
    if (0 ≤ tmp ≤ RW-1) // csn matches nr+tmp
      and (rbuff[nr-nd+tmp] is empty)
    then { rbuff[nr-nd+tmp] := msg.data };
    if tmp=0 then
      while (rbuff[nr-nd] is not empty) do nr := nr+1
  }

  progress-assumptions { weak fairness for Sendack and Deliver }
} // SWSink

```

Figure 3.4: Sink entity of the sliding window protocol.

Both entities execute all locally events with weak fairness. The source program allows any unacked data block to be repeatedly sent (in practice, the enabling condition of `Send()` and `Resend(j)` would be constrained by a congestion control scheme). Similarly, the sink program allows acks to be repeatedly sent (in practice, this would be constrained so that, for example, an ack is sent only in response to a received data message).

We show in the next section that if $N \geq SW + RW$ holds, then the sliding window protocol system offers fifo channel service from x to y using the lossy channel services between x and y . Formally,

Theorem 3.1 The composite system $\{SWSrc(x, y, N, Data), SWSink(y, x, N, Data)\}$ with $N \geq SW + RW$ offers $FifoChannel(x, y, Data)$ using $\{LossyChannel(x, y, DMsgs), LossyChannel(y, x, AMsgs)\}$.

3.4 Proof of service satisfaction

We prove theorem 3.1 in this section. The first step is to define the composite system of the entities and the services appropriately modified. We use the following notation for brevity.

- Src denotes $SWSrc(x, y, N, Data)$. Snk denotes $SWSnk(x, y, N, Data)$.
- Fxy denotes $FifoChannel(x, y, Data)$ -wrt- $\{Src, Snk\}$, that is, $FifoChannel(x, y, Data)$ modified to interact

with $\{\text{Src}, \text{Snk}\}$. [In particular, change $\text{Tx}(d)$ to an lc event and add a call to $\text{Src.Accept}(d)$ in its action; and change $\text{Rx}(d)$ to an xc event (to be called by $\text{Snk.Deliver}()$.)]

- Lxy denotes $\text{LossyChannel}(x, y, \text{DMsgs})\text{-wrt-}\{\text{Src}, \text{Snk}\}$. [In particular, modify $\text{LossyChannel}(x, y, \text{DMsgs})$ as follows: change $\text{Rx}(\text{msg})$ to an lc event and add a call to $\text{Snk.Receive}(\text{msg.data}, \text{msg.cj})$ in its action; and change $\text{Tx}(\text{msg})$ to an xc event (to be called by $\text{Snk.Sendack}()$.)]
- Lyx denotes $\text{LossyChannel}(y, x, \text{DMsgs})\text{-wrt-}\{\text{Src}, \text{Snk}\}$ (analogous to Lxy).
- Z denotes the (composite) system $\{\text{Fxy}, \text{Src}, \text{Snk}, \text{Lxy}, \text{Lyx}\}$.
- The “input” service events refer to the dnw events of Fxy and the upw events of Lxy and Lyx ; i.e., the events $\text{Fxy.Tx}(d)$, $\text{Lyx.Rx}(\text{msg})$, and $\text{Lxy.Rx}(\text{msg})$.
- The “output” service events refer to the upw events of Fxy and the dnw events of Lxy and Lyx ; i.e., the events $\text{Fxy.Rx}(d)$, $\text{Lyx.Tx}(\text{msg})$, and $\text{Lxy.Tx}(\text{msg})$.

Our task is to establish that the following conditions hold given $N \geq SW + RW$:

- Fault freedom: System $\{\text{Src}, \text{Snk}\}$ is fault-free in isolation.
- Input safety: For each input service event e , there is a matching xc event f in $\{\text{Src}, \text{Snk}\}$ and Z satisfies $\square(e.\text{ec} \Rightarrow f.\text{ec})$.
- Output safety: For each output service event e , if lc event f of $\{\text{Src}, \text{Snk}\}$ calls e , then Z satisfies $\square(f.\text{ec} \Rightarrow e.\text{ec})$.
- Progress: Z satisfies progress obligations of Fxy assuming the progress obligations of Lxy and Lyx .

Proof of fault-freedom

Because Src and Snk do not directly interact, system $\{\text{Src}, \text{Snk}\}$ is fault free iff Src and Snk are each fault-free in isolation. The only way for Src to become faulty is to execute an event that references $\text{sbuff}[i]$ for some i not in sbuff 's domain. But this is precluded because the following predicates are invariant (each A_i holds initially and is preserved by each Src event):

$$A_1 : \text{sbuff.domain} = [0..ng - na - 1]$$

$$A_2 : 0 \leq na \leq ns \leq ng$$

The only way for Snk to become faulty is to execute an event that references $\text{rbuff}[i]$ for some i not in rbuff 's domain. This is precluded because the following predicates are invariant:

$$A_3 : \text{rbuff.domain} = [0..RW - 1]$$

$$A_4 : 0 \leq nd \leq nr \leq nd + RW$$

Proof of input safety

The input service events are $\text{Fxy.Tx}(d)$, $\text{Lyx.Rx}(\text{msg})$, and $\text{Lxy.Rx}(\text{msg})$. $\text{Fxy.Tx}(d)$ is mapped to $\text{Src.Accept}(d)$, which matches (in parameter and type) and is always enabled. $\text{Lyx.Rx}(\text{msg})$ is mapped to $\text{Src.Rec}(\text{msg.cj})$, which matches and is always enabled. $\text{Lxy.Rx}(\text{msg})$ is mapped to $\text{Snk.Rec}(\text{msg.data}, \text{msg.cj})$, which matches and is always enabled.

Proof of output safety

The output service events are $Fxy.Rx(d)$, $Lyx.Tx(msg)$, and $Lxy.Tx(msg)$. $Snk.Deliver()$ calls $Fxy.Rx(d)$, and the call matches in type. $Snk.Sendack()$ calls $Lyx.Tx(msg)$, and the call matches in type. $Src.Send()$ and $Src.Resend(j)$ call $Lxy.Tx(msg)$, and each call matches in type. Because $Lyx.Tx(msg)$ and $Lxy.Tx(msg)$ are always enabled, there is nothing more to establish for these events. So all that remains is to establish that $Fxy.Rx(d)$ is enabled when it is called, and this is where all the interesting stuff happens.

$Snk.Deliver()$ calls $Fxy.Rx(rbuff[0])$, so we need to show that the latter is enabled whenever the former is enabled. Formally, we need to show that Z satisfies $\Box B_1$, where

$$B_1 : (nd < nr) \Rightarrow ((Fxy.rhx \circ \langle rbuff[0] \rangle) \text{ prefix-of } Fxy.txh)$$

To establish B_1 , we start by identifying properties we expect to hold. We expect nd to indicate the size of $Fxy.rhx$, and ng to indicate the size of $Fxy.txh$. Because $rbuff$ is intended to buffer data blocks nd to $nd + RW - 1$, we expect $rbuff[i]$ to be either empty or equal to $Fxy.txh[nd + i]$. Also, we expect $rbuff[i]$ to be non-empty for $nd + 1 < nr$ and empty for $nd + i \geq ns$ (since these latter data blocks have not yet been sent). We expect na to be at most nr , since nr has not yet been acked. We expect nr to be at most ns , since ns has not yet been sent. We expect ns to be at most $na + SW$, because Src has at most SW outstanding data blocks. Summarizing, we expect Z to satisfy $\Box B_{2..6}$, where:

$$B_2 : (nd = Fxy.rhx.size) \text{ and } (ng = Fxy.txh.size)$$

$$B_3 : [\text{forall } 0..RW-1 \ i :: (rbuff[i] \text{ is empty}) \text{ or } (rbuff[i] = Fxy.txh[nd+i])]$$

$$B_4 : [\text{forall } 0..RW-1 \ i, \ nd+i < nr :: rbuff[i] \text{ is non-empty}]$$

$$B_5 : [\text{forall } 0..RW-1 \ i, \ nd+i \geq ns :: rbuff[i] \text{ is empty}]$$

$$B_6 : na \leq nr \leq ns \leq na + SW$$

Note that $B_{2..4} \wedge A_{3..4}$ implies the consequent of B_1 . So it suffices to establish $\Box B_{2..6}$. It is easy to see that the latter holds if each received cyclic sequence number is correctly interpreted by the receiver. We now obtain the minimum value of N that ensures correct operation. For analysis, let us assume that each message contains the unbounded sequence number corresponding to the modulo- N sequence number. So a (data or ack) message j means a message with unbounded sequence number j .

Consider what happens when a data message j is received. Entity y , which has access to $\text{mod}(j, N)$ only, looks for a match in the receive window $nr, \dots, nr + RW - 1$. We want the following to be invariant, where $Lxy.transit(j)$ stands for $Lxy.rhx \circ \langle \text{data } j \rangle$ subsequence-of $Lxy.txh$:

$$(Lxy.transit(j) \text{ and } (\text{mod}(j - nr, N) \leq RW - 1)) \Rightarrow (j = \text{mod}(j - nr, N) + nr)$$

What values of j ensure the above? If j lies in the receive window, it is correctly matched. If j is very much lower than nr or very much higher than $nr + rw$, then $\text{mod}(j, N)$ wraps around and incorrectly matches a number in the receive window. Specifically, for decreasing j starting from nr (i.e., $j = nr - 1, nr - 2, \dots$), the first number that is incorrectly matched is $nr + RW - 1 - N$ (which gets mapped to $nr + RW - 1$). So we want $j \geq nr + RW - N$. Similarly, for increasing j starting from $nr + RW - 1$, the first value that is incorrectly matched is $nr + N$ (which gets mapped to nr). So we also want $j \leq nr + N - 1$. Combining the two bounds, we want the following to be invariant:

$$C_1 : Lxy.transit(j) \Rightarrow nr - N + RW \leq j \leq nr + N - 1$$

Similarly, when an ack message j is received, entity x , which has access to $\text{mod}(j, N)$ only, looks for a match in the range $na + 1, \dots, ns$. We want the following to be invariant, where $Lyx.transit(j)$ stands for $Lyx.rhx \circ \langle \text{ack } j \rangle$ subsequence-of $Lyx.txh$:

$$(Lyx.transit(j) \text{ and } (1 \leq \text{mod}(j - na, N) \leq ns - na)) \Rightarrow (j = \text{mod}(j - na, N) + na)$$

If j is in the range $na + 1, \dots, ns$. it is correctly matched. As j gets away from this range, the first value that is incorrectly matched is $na + N + 1$ on the upper side and $ns - N$ on the lower side. So it suffices if the following predicate is invariant:

$C_2 : \text{Lxy.transit}(j) \Rightarrow ns - N + 1 \leq j \leq na + N$

If C_1 and C_2 are kept invariant, then received cyclic sequence numbers are correctly matched and the protocol satisfies $\square B_{2..6}$ (in terms of proof rules, $A_{1..4} \wedge B_{2..6}$ is preserved by every event assuming $C_{1,2}$ holds prior to the event).

So what remains is to ensure that C_1 and C_2 are invariant. We next give an operational proof that $N \geq SW + RW$ suffices for this. Below, for a variable x , we use $x(t)$ to denote the value of x at time t .

C_1 can be falsified in only two ways. One is the transmission of a data message k . Because k lies in $na, \dots, ns - 1$, this preserves C_1 if $na \geq nr - N + RW$ and $ns - 1 \leq nr + N - 1$ hold. The first bound, $na \geq nr - N + RW$, holds if $na \geq ns - N + RW$ (because $ns \geq nr$), which holds if $na \geq na - N + RW + SW$ (because $ns \geq na - SW$), which holds if $N \geq SW + RW$. The second bound, $ns - 1 \leq nr + N - 1$, holds if $ns \leq na + N$ (because $nr \leq ns$), which holds because $ns \leq na + SW$ and $SW < N$.

C_1 can also be falsified by a data message reception that increases nr to an extent that violates the lower bound for some data message j in transit. Let k_1 and k_2 be the values of nr before and after the data message reception, at say time t_1 ; so the received data block was k_1 and prior to t_1 data blocks $k_1 + 1, \dots, k_2 - 1$ were buffered. So data block $k_2 - 1$ was sent at some earlier time, say $t_2 (< t_1)$. At that time, the value of ns , denoted by $ns(t_2)$, satisfied $ns(t_2) \geq k_2$. So $na(t_2) \geq k_2 - SW$, which implies $na(t_2) \geq k_2 - N + RW$ (because $N \geq SW + RW$). This means that any data message j sent after time t_2 , including those in transit at time t_1 , satisfies $j \geq k_2 - N + RW$. So C_1 is preserved.

C_2 's invariance is simpler to argue. Because nr never decreases, successive ack transmissions have non-decreasing unbounded sequence numbers. Because acks in transit are never reordered, the sequence of j 's in transit is non-decreasing and upper bounded by nr . Because na is the highest j received thus far, every ack message k in transit is lower bounded by na . Thus every ack message j in transit satisfies $na \leq j \leq nr$, which implies C_2 .

Assertional proof of $C_{1,2}$'s invariance We obtain assertional versions of the above operational arguments. Define the following predicates, where $\text{Lxy.transit}(k, j)$ stands for $\text{Lxy.rhx} \circ \langle \text{data } k, \text{data } j \rangle$ subsequence-of Lxy.txh :

$D_1 : \text{Lxy.transit}(j) \Rightarrow j \leq ns - 1$

$D_2 : \text{Lxy.transit}(j) \Rightarrow j \geq nr - SW$

$D_3 : (\text{Lxy.transit}(j) \text{ and } \text{rbuff}[i] \neq \text{empty}) \Rightarrow j \geq nd + i - SW + 1$

$D_4 : \text{Lxy.transit}(k, j) \Rightarrow j \geq k - SW + 1$

$\square D_{1..4}$ holds because $D_{1..4}$ satisfies invariance rule assuming $B_{1..6}$ [in particular: data send preserves D_1 due to B_6 , D_2 due to B_6 , D_3 due to $B_{5,6}$, D_4 due to $D_{1,2}$ AND B_6 ; data reception preserves D_2 due to D_3 , D_3 due to D_4]. $\square C_1$ holds because $(D_1 \wedge D_2 \wedge B_6) \Rightarrow C_1$ holds.

We now turn to C_2 . Define the following predicates, where $\text{Lxy.transit}(j, k)$ stands for $\text{Lxy.rhx} \circ \langle \text{ack } k, \text{ack } j \rangle$ subsequence-of Lxy.txh :

$E_1 : \text{Lxy.transit}(j) \Rightarrow j \geq na$

$E_2 : \text{Lxy.transit}(j, k) \Rightarrow k \geq j$

$\square E_{1,2}$ holds because $E_{1,2}$ satisfies invariance rule assuming $B_{1..6}$. $\square C_2$ holds because $(E_1 \wedge E_2 \wedge B_6) \Rightarrow C_2$ holds.

Proof of progress

We need to show that Z satisfies Fxy 's progress obligation assuming the progress obligations of Lxy and Lyx . Fxy 's progress obligation, given B_2 , can be stated as $ng \geq k \rightsquigarrow nd \geq k$. Because $nr \geq k \rightsquigarrow nd \geq k$ holds (via $\text{Deliver}()$), it suffices to establish $ng \geq k \rightsquigarrow nr \geq k$.

We first establish the following:

$P_1 : ns \geq k \rightsquigarrow nr \geq k$ (via Resend() and Lxy progress)

$P_2 : nr \geq k \rightsquigarrow na \geq k$ (via Sendack() and Lyx progress)

$P_3 : ns \geq k \rightsquigarrow na \geq k$ (P_1, P_2)

Because of P_1 , the desired assertion, $ng \geq k \rightsquigarrow nr \geq k$, is implied by $ng \geq k \rightsquigarrow ns \geq k$, which is implied by $ng > ns = j \rightsquigarrow ns > j$. The last assertion holds via Send() if $j + 1$ is in the send window. Otherwise, P_3 tells us that na increases, after which $j + 1$ enters the send window. In summary:

$P_4 : ((ng > ns = j) \wedge (j < na + SW)) \rightsquigarrow (ns > j)$ (via Send())

$P_5 : ((ng > ns = j) \wedge (j = na + SW)) \rightsquigarrow (na \geq j)$ (P_3)

P_4 and P_5 (with B_6) imply the desired result, $(ng > ns = j) \rightsquigarrow (ns > j)$. This completes the progress proof, and the proof of theorem 3.1.

3.5 Offering fifo channels using LRDml channels

The above distributed system does not offer a fifo channel using LRD channels. This is because, for any given N , a channel that can reorder and duplicate messages without limits can always present to the receiver a message that will be incorrectly matched (e.g., the data message 0 when nr equals N). But the system does offer the fifo channel using LRDml channels if successive increments of ns are at least δ seconds apart and

$$N \geq SW + RW + \frac{L}{\delta}$$

where L is the maximum message lifetime of the LRDml channels. Formally,

Theorem 3.2 Let $SWSrcT(x, y, N, Data)$ be $SWSrc(x, y, N, Data)$ in which executions of Send() are at least δ seconds apart. The composite system $\{SWSrcT(x, y, N, Data), SWSink(y, x, N, Data)\}$ with $N \geq SW + RW + L/\delta$ offers $FifoChannel(x, y, Data)$ using $\{LRDmlChannel(x, y, DMsgs, L), LRDmlChannel(y, x, AMsgs, L)\}$.

The proof of theorem 3.2 is exactly like the proof of theorem 3.1, except that the composite system Z has real-time features, and the proof of $\square C_{1,2}$ is different.

Modifications to Z. Z differs from that for lossy channels as follows. Lossy channels are replaced by LRDml channels. Include the RealTime program in Z . To capture the δ time constraint, introduce another real-time variable, say τns , include $\tau ns := \tau now$ in Send()'s action, and include $\tau now - \tau ns > \delta$ in Send()'s enabling condition.

The proof that Z satisfies the four conditions (fault-freedom, input safety, output safety, progress) is exactly the same as in the lossy case, except for that of $\square C_{1,2}$, whose proof is next.

C_1 's upper bound holds exactly as in the lossy channel case. Now consider C_1 's lower bound. At time t_0 , let (j, t) be in $Lxy.txh$ such that $t > 0 - L$. We want $j > nr(t_0) - N + RW$ to hold. It suffices if $j > ns(t_0) - N + RW$ holds (because $nr(t_0)$ is at most $ns(t_0)$). At time t , when data j was sent, j was at least $na(t)$ (otherwise j would not have been sent), which was at least $ns(t) - SW$. During the interval $[t, t_0]$, ns has increased by at most L/δ . So $j \geq ns(t_0) - L/\delta - SW$. So the desired bound of $j > ns(t_0) - N + RW$ holds if $SW + L/\delta \leq N - RW$, or equivalently, $N \geq SW + RW + L/\delta$.

C_2 's upper bound holds exactly as in the lossy channel case. Now consider C_2 's lower bound. At time t_0 , let (j, t) be in $Lyx.txh$ such that $t > 0 - L$. We need to show that $j \geq ns(t_0) - N + 1$ holds. At time t , when the ack j was sent, j equalled $nr(t)$, which was at least $na(t)$, which was at least $ns(t) - SW$. During the interval $[t, t_0]$, ns has increased by at most L/δ . So $j \geq ns(t_0) - L/\delta - SW$, which implies the desired bound given $N > SW + L/\delta$.

Assertional proof of $C_{1,2}$'s invariance For assertional versions of the above operational arguments, we first introduce some auxiliary variables. Define array ts such that $ts[k]$ indicates the time when ns exceeded k . For notational convenience, let ts be an infinite array indexed over non-negative integers, and let $ts[k]$ equal ∞ initially, where ∞ satisfies $\infty > k$ and $\infty + k = \infty$ for any integer k . Arrays ta and tr are similarly defined for na and nr , respectively.

Formally, insert the following in Z :

```

real[0..] ts := ∞           // in Src.init( )
real[0..] ta := ∞           // in Src.init( )
ts[ns-1] := τnow;          // in Src.Send(d).ac, at the end
ta[na .. na+tmp-1] := τnow; // in Src.Receive(msg).ac, just before na := na+tmp
real[0..] tr := ∞;          // in Snk.init( )
tr[nr] := τnow;            // in Snk.Send(d), just before nr := nr+1

```

For C_1 , define the following predicates:

$$F_1 : (ta[0] \leq ta[1] \leq \dots \leq ta[na-1] \leq \tau\text{now}) \text{ and } [\forall j, j \geq na :: ta[j] = \infty]$$

$$F_2 : (ts[0] \leq ts[1] \leq \dots \leq ts[ns-1] \leq \tau\text{now}) \text{ and } [\forall j, j \geq ns :: ts[j] = \infty]$$

$$F_3 : [\forall j, j \geq 0 :: ta[j] \geq ts[j]]$$

$$F_4 : [\forall j, j \geq 0 :: ts[j + SW] \geq ta[j]]$$

$$F_5 : (((j, t) \in Lxy.txh) \text{ and } t > ta[k]) \Rightarrow j > k$$

$$F_6 : [\forall j, j \in [0..ns-2] :: ts[j+1] \geq ts[j] + \delta]$$

$$F_7 : [\forall k, k \in [0..ns-1] :: ts[ns-1] \geq ts[ns-1-k] + k\delta]$$

$$F_8 : (((j, t) \in Lxy.txh) \text{ and } t > \tau\text{now} - L) \Rightarrow j \geq ns - SW - L/\delta$$

$\square F_{1..6}$ holds because $F_{1..6}$ satisfies the invariance rule assuming $B_{1..6}$; that is, $F_{1..6}$ holds initially, and $\{F_{1..6}\}e\{F_{1..6}\}$ holds, for every event, including Age. [In particular: each of F_1, F_2, F_3, F_4 is preserved by every event due to B_6 ; data send preserves F_5 is preserved by data send and by ack receive due to B_6 and $F_{1..4}$; F_6 is preserved by data send.] Note that $\square F_{1..5}$ does not require the δ constraint.

$\square F_7$ holds because $F_6 \Rightarrow F_7$ holds. $\square F_8$ holds because $F_{1..7} \Rightarrow F_8$ holds [(j, t) in $Lxy.txh$ and $t > \tau\text{now} - L$ imply $t > ts[ns-1] - L$ (from F_2), which implies $t > ts[ns-1 - L/\delta]$ (from F_7), which implies $t > ta[ns-1 - L/\delta - SW]$ (from F_4), which implies $j > ns - 1 - L/\delta - SW$ (from F_5), which implies $j \geq ns - L/\delta - SW$].

$\square C_1$ holds because $F_8 \Rightarrow C_1$ given $N \geq SW + RW + L/\delta$.

For C_2 , define the following predicates:

$$G_1 : (tr[0] \leq tr[1] \leq \dots \leq tr[nr-1] \leq \tau\text{now}) \text{ and } [\forall j, j \geq nr :: tr[j] = \infty]$$

$$G_2 : [\forall j, j \geq 0 :: ta[j] \geq tr[j]]$$

$$G_3 : (((j, t) \in Lyx.txh) \text{ and } t > tr[k]) \Rightarrow j > k$$

$$G_4 : (((j, t) \in Lyx.txh) \text{ and } t > \tau\text{now} - L) \Rightarrow j \geq ns - SW - L/\delta$$

$\square G_{1..3}$ holds because $G_{1..3}$ satisfies the invariance rule. $\square G_4$ holds because $(G_{1..3} \wedge F_{2,4}) \Rightarrow G_4$ holds (similarly to $F_{1..7} \Rightarrow F_8$). $\square C_2$ holds because $F_7 \Rightarrow C_2$ given $N \geq SW + RW + L/\delta$.

3.6 Concluding remarks

The above services and protocols can be extended in various ways. Regarding the service, all the channels specified here are non-blocking channels, that is, a message can be sent at any time. It is simple to add blocking, where a message can be sent only under certain conditions e.g., when the channel is not “full”, or the sender’s “quota” has not been exceeded, or by the offerer giving explicit permission for each transmission.

The latter is in fact what happens if user x executes a `send(m)` call that may block internally, returning only after the message m is accepted by entity x . In this case, the start of the call corresponds to our `dnw Tx(m)` event, the return corresponds to a new `upw` event, say `OkToTx()`, and `Tx(m)` is allowed only after an `OkToTx()` response to the previous `Tx(m)`.

A similar flow control can be imposed between entity y and user y . Consider the common case where user y executes a `receive()` call to get the next message. In this case, the start of the call corresponds to a new `dnw` event, say `OkToRx()`, the return corresponds to the `upw` event `Rx(m)`, and `Rx(m)` is allowed only after an `OkToRx()`.

Another type of extension is to add maximum delay constraints. For example, a maximum delay fifo channel delivers the message within D seconds of it being sent. A maximum delay lossy channel delivers a message only within D seconds of it being sent; if it is not delivered by then, it is lost.

The protocols also can be extended in various ways. One extension is to impose flow control between a user and its local entity, mirroring the flow controlled services described above. This would allow each entity to have only bounded amount of storage.

Other extensions can improve the performance of the protocol. The above protocol uses cumulative acknowledgments. We can also use “negative” acknowledgements (nacks) to indicate gaps in the received data. Nacks allow the data source to retransmit missing data sooner than cumulative acks. The protocol can also use “selective” acknowledgements (sacks) to indicate out-of-sequence data received. This allows the data source to retransmit only what is needed, rather than everything outstanding. Selective acks and nacks are not usually used in TCP, although they are available as options and there are studies indicating that they can improve performance significantly.

The above protocol uses fixed-size data blocks. An alternative is to send variable-sized data blocks. This can be done by augmenting the data messages with a field indicating the size of the data block. Another alternative is to send a variable number of data blocks in a message; if the data blocks are consecutive, the message needs only identify the sequence number of the first data block and the number of data blocks. TCP does the latter with an octet, or byte, as the data block size. A similar modification would be needed for selective and negative acks.

Flow-control can also be introduced between entity x and entity y , so that entity x does not send data faster than the entity y can handle. By dynamically varying the send window size, the sliding window mechanism can also achieve flow control. In particular, consumer-directed flow control works as follows: entity y regularly informs entity x of its current receive-window size (using say another field in acknowledgement messages), and entity x sets its send-window size accordingly. Note that in this case, the above condition on N reduces to $N \geq 2RW + L/\delta$.

3.7 Exercises

- 3.1 Change the above fifo channel service to include flow control between entity x and user x , entity y and user y , and entity y and entity x .
- 3.2 Change the above protocol to offer the fifo channel in problem 3.1, that is, include flow control between entity x and user x , entity y and user y , and entity y and entity x . Ensure that the resulting entities require a bounded amount of buffer.
- 3.3 In the protocol developed above, retransmissions are initiated by entity x and entity y only responds with acks. Modify the protocol so that retransmissions of ack messages are initiated by entity y and entity x sends data messages only in response.
- 3.4 Give an assertional proof of C_1 for the lossy channel case. That is, come up with a predicate C_3 such that C_3 holds initially, C_3 is preserved by every event, and $C_3 \Rightarrow C_1$. Repeat for C_2 .

3.5 Do problem 3.4 for the LRDml channel case.