

Chapter 3

Message-Passing Services Examples

A message-passing service, or **channel** for short, allows a set of users to send and receive messages. An example is the service provided by IP in the Internet. This chapter continues the informal introduction to SESF by specifying various kinds of channels. Distributed systems that offer and use these channels are described in the next chapter.

Channels have various attributes relevant to correctness. A channel can be reliable or unreliable. A reliable channel acts as a fifo buffer for every ordered pair of users, delivering messages in the order sent. Its progress obligation is that every message sent is eventually delivered. An unreliable channel can lose, reorder, duplicate, and/or corrupt messages in transit. Its progress obligation is that a message repeatedly sent is repeatedly delivered. We are usually interested in two kinds of unreliability: “lossy” and “LRD”. A lossy channel can only lose messages, while a LRD channel can lose, reorder, and duplicate messages in transit.

A channel may satisfy timing constraints between its sends and receptions, such as maximum message delay or maximum message lifetime. Traditionally, timing constraints have been ignored in correctness analysis, in that one strives for distributed systems that operate correctly regardless of real-time constraints. However, it is often the case in networking that real-time constraints are intrinsic to the correctness of a system. For example, reliable data transfer using cyclic sequence numbers is not possible over LRD channels unless there is a maximum message lifetime.

One often-overlooked distinction is whether or not the receive event explicitly identifies the sender of the received message, that is, does it pass up the received message and sender id, or only the received message. The latter is equivalent to the former if the message includes a sender id field *and the sender sets it properly*; in particular, the service does not prevent a sender from “spoofing” another sender. It is worth noting that this “anonymous sender” service is essentially what the IP layer in the Internet provides.

Another distinction is between “unicast” and “broadcast” service. In unicast service, a transmitted message is intended for a single receiver. In broadcast service, a transmitted message is intended for all other users or a static subset of users (e.g., within a subnet). Typically, broadcast service is added to unicast service by introducing a special broadcast id (e.g., the all 1’s address).

The following sections specify different kinds of channels. We start with point-to-point channels, that is, channels with one sender and one receiver. We cover fifo, lossy, and LRD types, for both time-unconstrained and time-constrained cases. Then we examine multi-user channels where the receive event does not identify the sender. Finally, we look at broadcast channels. We emphasize that our interest here is on correctness analysis. (For performance analysis, one would also need a probabilistic characterization of the channel delays, losses, and reordering.)

Throughout, `UserId` denotes the ids of the users of the channel, and `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).

The specifications involve sequences of messages and common operations on sequences, e.g., concatenation, prefix, subsequence, merge. We use “ \langle ” and “ \rangle ” to define sequences, for example, $\langle \rangle$ is the empty sequence and $\langle a, b, c \rangle$ is a three element sequence with `a` being the first and `c` the last. We use “ \circ ” for con-

catenation, so $\langle a \rangle \circ \langle b, c \rangle$ is equal to $\langle a, b, c \rangle$. 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. So a finite sequence $x = \langle x[0], x[1], \dots, x[x.size - 1] \rangle$.

3.1 Point-to-point channels

A point-to-point channel is a special case of a channel where there is one sender, identified as x below, and one receiver, identified as y below. Both x and y range over `UserId`.

Fifo channels. We start by specifying fifo channels.

```

service-program FifoPpChannel( 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 ]
  }
} // fifo point-to-point channel

```

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 not 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 LossyPpChannel( 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 ◦ ⟨ 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 ]
    ]
  }
} // LossyPpChannel

```

As in the fifo case, the service specification does not 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 $txh = \langle a, b, c, d \rangle$ and $rxh = \langle a, b \rangle$, 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 $rxh \circ \langle m \rangle$ subsequence-of txh holds.

LRD channels. A LRD channel service is obtained from the lossy channel service by changing the enabling condition of $Rx(m)$ 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.

3.2 Time-constrained point-to-point channels

We now specify real-time versions of fifo, lossy, and LRD channels, called fifo-md (fifo max delay), lossy-md (lossy max delay), and LRDml (LRD max lifetime) channels, respectively. A fifo-md channel is just a fifo channel except that every message sent is delivered within a specified max delay of transmission. A lossy-md channel is just like a lossy channel except that every message sent is delivered or lost within a specified max delay of transmission. A LRDml channel is like a LRD channel except that it has a maximum message lifetime (typically orders larger than the average delay). Within the maximum message lifetime of a message being sent, that message (and any duplicates) is no longer the channel and hence cannot be received at a later point in time.

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

```

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

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

```

A time-constrained system or service can now be modeled by using τ_{now} in event enabling conditions and actions (assume that τ_{now} is readable by all). Consider the constraint that executions of event e are separated by at least D seconds. To model this, introduce a variable, say τ_e , to record the time when e happens, add $\tau_e := \tau_{now}$ in e ’s action, and add $\tau_{now} < \tau_e + D$ as a conjunct in e ’s enabling condition. Now consider the constraint that executions of event e are separated by no more than D seconds. To model this, proceed as above except that instead of the conjunct $\tau_{now} < \tau_e + D$ in e ’s enabling condition put the

conjunct $\tau_{\text{now}} + \Delta < \tau_e + D$ in $\text{Age}(\Delta)$'s enabling condition. The former kind of constraint is a “delay” constraint and the latter kind is a “deadline” constraint.

Fifo-md channels. A fifo-md channel program of max delay D seconds is obtained by tweaking the fifo channel program as follows. First, let txh record the sequence of messages sent along with their transmit times, i.e., in $\text{T}\times(\text{msg})$, append $\langle \text{msg}, \tau_{\text{now}} \rangle$ to txh . Similarly, let rxh records the sequence of messages received along with their receive times. Finally, introduce the deadline constraint that every message in transit has to be younger than D seconds. Because the channel is fifo, $\text{txh}[\text{rxh.size}]$ is the next message to be received, and it is in transit iff $\text{txh.size} > \text{rxh.size}$ holds. The resulting service program is as follows, where for any entry x of txh or rxh , $x.\text{msg}$ denotes the message component and $x.\text{time}$ denotes the time value:

```

service-program FifoMdPpChannel( x, y, Msg, D ) { // fifo max delay D point-to-point channel
  MsgTimeSeq txh :=  $\langle \rangle$ ; // sequence of (message, time) pairs transmitted
  MsgTimeSeq rxh :=  $\langle \rangle$ ; // sequence of (message, time) pairs received

  dnw T $\times$ ( Msg msg ) {
    ec true
    ac append( txh,  $\langle \langle \text{msg}, \tau_{\text{now}} \rangle \rangle$  )
  }

  upw R $\times$ ( Msg msg ) {
    ec true
    ac append( rxh,  $\langle \langle \text{msg}, \tau_{\text{now}} \rangle \rangle$  )
  }

  deadline: {
    (txh.size > rxh.size)  $\Rightarrow$  ( $\tau_{\text{now}} - \text{txh}[\text{rxh.size}].\text{time} < D$ )
  }

  progress-obligation() {
    [  $\forall$  int i : txh.size  $\geq$  i  $\rightsquigarrow$  rxh.size  $\geq$  i ]
  }
} // fifo-md channel

```

The acceptable sequences of event calls defined by this service are those that are generated by the above program executing in conjunction with program `RealTime`.

Lossy-ml channels. The lossy-ml channel is a lossy channel in which a message is received only if its receive time is not more than L plus its transmit time. Note that this is a delay, rather than a deadline, constraint. The resulting service program is the same as for `fifo-md` except for two changes. First, the progress obligations becomes that of `LossyPpChannel`, i.e., a message repeatedly sent is repeatedly sent. Second, the enabling condition of $\text{R}\times(\text{m})$ changes to the following, where for any sequence x of message-time pairs, $x.\text{msg}$ denotes the sequence of messages in x and $x.\text{time}$ denotes the sequence of time values in x :

```

[for some MsgSeq x:: x subsequence-of txh and (rxh.msg  $\circ$   $\langle \text{m} \rangle$ ) = x.msg
  and [for all int i, 0  $\leq$  i < rxh.size:: rxh[i].time < x[i].time + L ]
  and  $\tau_{\text{now}} < x[\text{rxh.size}].\text{time} + L$  ]

```

LRD max lifetime channel. The `LRDml` point-to-point channel with maximum message lifetime L is similar to the lossy-ml case except that the enabling condition of $\text{R}\times(\text{m})$ changes to $((\text{m}, \text{t}) \text{in txh})$ and $(\tau_{\text{now}} < \text{t} + L)$, so that only messages younger than L seconds are received.

3.3 Multi-user channels

A point-to-point channel has one sender and one receiver. A multi-user channel has a set of users, each of which be a sender and a receiver. Clearly, a multi-user channel can be defined as a set of point-to-point channels, one for every ordered pair of users. In this case, every receive event identifies the sender of the received message. In this section, we look at the more interesting “anonymous sender” case. Here, the send event signature identifies both the sender and the destination, but the receive event signature identifies only the receiver. So when a user receives a message, it does not learn of the sender’s identity (unless the sender puts its identify in the message).

Fifo channels. We first define a multi-user channel where message delivery between any ordered pair of users is fifo.

```

service-program FifoChannel( UserId, Msg ) { // multi-user fifo channels
  MsgSeq[UserId, UserId] txh := ⟨⟩;
  MsgSeq[UserId] rxh := ⟨⟩;
  // txh[x,y] is the sequence of messages sent by x destined for y
  // rxh[x] is the sequence of messages received by x

  dnw Tx( UserId x, y, Msg msg ) {
    ec true
    ac append( txh[x,y], msg )
  }

  upw Rx( UserId x, Msg msg ) {
    ec ( rxh[x] ◦⟨ msg ⟩ ) prefix-of merge-of({txh[y,x]:: UserId y})
    ac append( rxh[x], msg )
  }

  progress-obligation() {
    [ ∀ UserId x, int i : (∑y txh[x,y].size) ≥ i ↔ rxh[x].size ≥ i ]
  }
} // multi-user fifo channel

```

Note that even though the service is fifo between every pair of users, we need to resort to the merge operation to allow for any all possible interleavings of receives at a node. This complexity would not arise if the receive event’s signature identifies the message sender’s id, because then each receiver has a separate receive event for each sender.

Another way to overcome this complexity is to allow internal nondeterminism in the service. For example, we can allow the receive event at y to non-deterministically choose a sender x with nonempty $txh[x,y]$ to receive from. This non-determinism is internal because the choice does not affect the signature of the receive event. We do not choose this approach because (1) services with internal nondeterminism are harder to understand in general, although perhaps not in this case, and (2) it greatly complicates the program-based notion of satisfaction.

Another point to note is that the progress obligation above allows a transmitted message to be never received if other users keep sending messages to the same destination. This is weaker than the progress of the point-to-point service. The stronger requirement, that every sent message is eventually received, can be easily captured. For example, tag every transmitted message with a unique identifier, say sequence number and sender id, and require every transmitted identifier to be eventually received.

Lossy and LRD channels. The multi-user lossy channel is obtained by modifying the above fifo channel 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:

```
[  $\forall$  UserId x, y, Msg m:
  [  $\forall$  Nat i: nbr(txh[x,y], m)=i  $\rightsquigarrow$  nbr(txh[x,y], m)>i ]
   $\Rightarrow$  [  $\forall$  Nat i: nbr(rxh, m)=i  $\rightsquigarrow$  nbr(rxh, m)>i ]
]
```

The specification for LRD channels is the same except that the enabling condition of the Rx event simplifies dramatically to (msg in {txh[y,x] :: UserId y}).

3.4 Broadcast channels

In a broadcast channel, every user is a potential sender and every message sent is intended to be received by all users other than the sender. We first defines a broadcast service where message delivery between any ordered pair of users is fifo.

```
service-program FifoBroadcastChannel( Msg, UserId ) {
  MsgSeq[UserId] txh, rxh :=  $\langle \rangle$ ;
  // txh[x] is the sequence of messages transmitted by x
  // rxh[x] is the sequence of messages received by x

  dnw Tx( UserId x, Msg msg ) {
    ec true
    ac append( txh[x], msg )
  }

  upw Rx( UserId x, Msg msg ) {
    ec ( rxh[x] o $\langle$  msg  $\rangle$  ) prefix-of merge-of({txh[y,x]:: UserId y})
    ac append( rxh[x], msg )
  }

  progress-obligation() {
    [  $\forall$  UserId x, int i : ( $\sum_y$  txh[x,y].size) $\geq$ i  $\rightsquigarrow$  rxh[x].size $\geq$ i ]
  }
} // fifo broadcast service
```

Lossy broadcast and LRD broadcast can be obtained from fifo broadcast in the same way as lossy shared and LRD shared were obtained from fifo shared. This is left as an exercise.

3.5 Concluding remarks

The services specified here can be extended in various ways. One practically important extension would be that of flow control. 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 in the usual “sockets” interface, where, for example, user x makes 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()`.

We emphasize that kinds of channels considered here are adequate for the lower layers of networking, i.e., the data link and network layers (e.g., LAP, Ethernet, 802.11 (WiFi), IP). At higher layers of networking,

message-passing services do more than just message passing, and additional attributes need to be modeled. One such attribute is whether the channel is “connection-less” or “connection-based”. In a connection-less service, a sender can send without first establishing a connection to the destination. In a connection-based service, a sender can send only after establishing a connection to the destination. The latter is almost the case when some guaranteed level of quality of service is needed, because it gives the offerer an opportunity to allocate resources (at the end nodes and perhaps also at any intermediate nodes) before allowing the sender to send. In the Internet for example, TCP offers connection-based reliable delivery whereas UDP offers connection-less unreliable delivery. Multicasting is another such attribute. A multicast service is a generalization of broadcast service, in that the intended receivers for a message transmission is a dynamic subset of the users,

3.6 Exercises

- 3.1 Change the fifo point-to-point channel to include flow control between entity x and user x, entity y and user y, and entity y and entity x.
- 3.2 Atomic broadcast is a compromise between fifo broadcast and lossy broadcast. It is a lossy broadcast in which all nodes receive the same sequence of messages; i.e., a transmitted message is received by a user only if it is received (or will be received) at every other user. Specify the atomic broadcast service by filling in the boxes below (you can leave a box empty).

```

service-program AtomicBroadcastChannel( Msg, UserId ) {
  MsgSeq[UserId] txh, rxh := ⟨⟩;
  // txh[x] is the sequence of messages transmitted by x
  // rxh[x] is the sequence of messages received by x

  

  dnw Tx( UserId x, Msg msg ) {
    ec true
    ac append( txh[x], msg )
    
  }

  upw Rx( UserId x, Msg msg ) {
    ec 
    ac append( rxh[x], msg )
    
  }

  progress-obligation() {
    
  }
} // atomic broadcast service

```