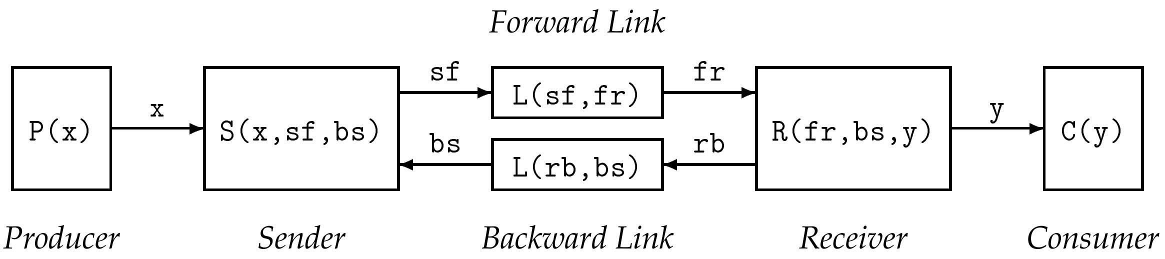

Figure 1: Connection from P to C via two L-links and protocol handlers S and R

Faulty Links

This task is about the communication of messages from a producer to a consumer (see Figure 1). A message is either a bit in $\{0,1\}$ or an error indicator E.

Producer P produces a sequence of messages on its output channel x (see Figure 2). Consumer C consumes a sequence of messages from its input channel y (see Figure 3). Let us denote the produced sequence by p_i, where $p_i \in \{0,1\}$ for $0 \leq i$, and the consumed sequence by c_i ($0 \leq i$).

We would like to guarantee that C consumes exactly the sequence produced by P, that is, $p_i = c_i$ for $0 \leq i$. In particular, *every bit sent by P should arrive at C exactly once in a finite number of steps*. One solution is obtained by connecting P's output channel directly to C's input channel (see Figure 4). In this task, however, P and C are far apart and, unfortunately, we have to connect them by faulty communication links.

A faulty communication link L, from now on called a link for short, has an input channel x and an output channel y (see Figure 5). It copies each message received on x to y, but it may at times damage messages. Damage is detectable: a damaged message has a special appearance, distinct from undamaged messages: $E \notin \{0,1\}$ (error).

A link that damages all messages from a certain moment onward would be useless for our purposes, because then it cannot be guaranteed that bits always arrive in a finite number of steps. We postulate that a link does not behave this way: a link will damage only a *finite* number of consecutive messages (see Figure 5). To put it differently: when a message is repeated often enough it is eventually transmitted correctly.

Let us connect P's output channel to the input channel of a link L, and L's output channel to C's input channel (see Figure 6). This configuration need not work as desired, because messages may be damaged and hence lost. For example, we can have $p_0 = 0$ and $c_0 = E$.

Retransmission Protocol

We can prevent the loss of messages by retransmitting damaged messages. For this purpose we also need a link in the opposite direction, called the backward link; the other link is called the forward link. The retransmission protocol is implemented by sender S on P's side and receiver R on C's side. S and R are connected to each other by the forward and backward links (see Figure 1).

Sender S (see Figure 7) inputs a message from P and sends it via the forward link to R. S then waits for an acknowledge bit from R via the backward link. Bit 1 indicates correct reception, bit 0 damaged reception. S will repeatedly retransmit the message until a 1 is received, after which the next message from P is processed.

Receiver R (see Figure 8) inputs a message from the forward link. When a damaged message is received, R requests a retransmission by sending a 0 to S via the backward link. Once an undamaged message is received, R outputs the message to C and sends a 1 along the backward link to S, after which processing continues with the next message.

Figure 9 shows a scenario for the transmission of $p_0 = 0$ and $p_1 = 1$, where transmission of p_0 involves a damaged message. The left-hand column, labeled T, shows the time steps. The other columns are labeled with channel identifiers and show the values sent along these channels.

Subtask A

The above protocol does not always work as desired. Give a scenario showing this. Indicate clearly on the accompanying Scenario Form A which values are communicated at what moments along which channels. You need not use use all time slots shown.

Subtask B

We change the programs for sender S and receiver R to improve the protocol. Sender S (see Figure 10) now tags each bit received from P alternatingly by 0 and 1. Otherwise, S works as before. A message $m \in$ `TMsg` with $m \neq E$ is a tagged bit: $m.v$ is the bit value and $m.t$ the tag.

Receiver R (see Figure 11) now has a local variable $u \in \{0,1\}$ indicating the tag it expects. Upon reception of a message via the forward link, the receiver repeats the following. When the message is damaged, R sends back 0. Otherwise, when the tag differs from u, it sends back 1 but does not deliver m to C. Only when an undamaged messaged with the expected tag arrives, is it delivered to C and is the expected tag updated. After all cases, R awaits a new message.

The programs for P, L, and C, and the way they are connected remain the same, except that the forward link now deals with messages that are tagged bits (`TMsg` instead of `Msg`).

Give a scenario showing that the improved protocol does not work as desired. Indicate clearly on the accompanying Scenario Form B which values are communicated at what moments along which channels. You need not use all time slots shown. Use an arrow to indicate a repetition of events.

Subtask C

Change sender S and receiver R to make the protocol work correctly. Write your answer on the Program Forms. You need not fill all empty lines.

You are only allowed to change the program text of S and R appearing between '`|`' and '`end`'. You are *NOT* allowed to add variables to S or R, or to change P, L, or C, or to change the way things are connected. You are also *NOT* allowed to invent new language constructs (statements or expressions).

```
P = proc (x!Msg).
begin i: var Nat & m: var Msg
| i := 0
; forever
  do m := p_i ; x!m
   ; i := i+1
  od
end
```

Figure 2: Program for producer P

```
C = proc (y?Msg).
begin i: var Nat & m: var Msg
| i := 0
; forever
  do y?m ; c_i := m
   ; i := i+1
  od
end
```

Figure 3: Program for consumer C

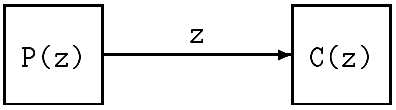

Figure 4: Direct connection from P to C

```
L = proc (x?Msg & y!Msg).
begin m: var Msg & n: var Nat
| forever
  do n :∈ Nat
   ; for n do x?m ; y!E od
   ; x?m ; y!m
  od
end
```

Figure 5: Program for faulty link L

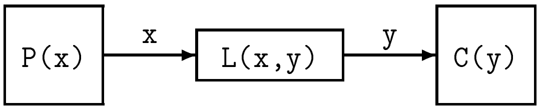

Figure 6: Connection from P to C via L

```
S = proc (x?Msg & sf!Msg & bs?Msg).
begin m, a: var Msg
| x?m
; forever
  do sf!m ; bs?a
   ; if a = 1 then x?m fi
  od
end
```

Figure 7: Program 1 for sender S

```
R = proc (fr?Msg & rb!Msg & y!Msg).
begin m: var Msg
| forever
  do fr?m
   ; if m = E then rb!0
     else y!m ; rb!1
     fi
  od
end
```

Figure 8: Program 1 for receiver R

T	*x*	*sf*	*fr*	*rb*	*bs*	*y*
0	0					
1		0				
2			*E*			
3				0		
4					0	
5		0				
6			0			
7						0
8				1		
9					1	
10	1					
11		1				
12			1			
13						1
T	*x*	*sf*	*fr*	*rb*	*bs*	*y*

Figure 9: Scenario for $p_0 = 0$ and $p_1 = 1$ involving a retransmission for p_0

```
S = proc (x?Msg & sf!TMsg & bs?Msg).
begin m: var TMsg & a: var Msg
| m.t := 0 ; x?m.v
; forever
  do sf!m ; bs?a
   ; if a = 1
     then m.t := 1-m.t ; x?m.v
     fi
  od
end
```

Figure 10: Program 2 for sender *S*

```
R = proc (fr?TMsg & rb!Msg & y!Msg).
begin m: var TMsg & u: var Nat
| u := 0
; forever
  do fr?m
   ; if m = E then rb!0
     else if m.t = u
          then y!m.v ; u := 1-u
          fi
        ; rb!1
     fi
  od
end
```

Figure 11: Program 2 for receiver *R*