PROTOCOL DESIGN 7

7.1 INTRODUCTION

So far, our discussion of protocol design has covered specification and structuring methods, design principles, and solutions to a set of standard protocol problems. It is time to see how we can apply all this to a real design problem. As an example, in this chapter we undertake the design of a protocol for the transfer of files between two asynchronous machines.

Our purpose in this chapter is to use a disciplined method to specify the five essential elements of the protocol (Chapter 2, Section 2.2). For the protocol procedure rules our aim is to construct a high level prototype with explicit correctness criteria. Later, we will then be able to show convincingly that the design criteria are either satisfied or violated, using automated tools.

We will try to do this by applying the rules of design that are listed in Chapter 2, Section 2.8. Most important among those for validation purposes is Rule 7:

> *Before implementing a design, build a high level prototype and verify that the design criteria are met.*

Our prototype is a validation model, as defined in Chapters 5 and 6. A verification that the design criteria are met is done in Chapter 14, with the tools that are developed in Chapters 11 to 13.

We make two crucial assumptions about the design process.

- Protocol design is an iterative process. The design is not likely to be correct the first time it is written down, and very likely it will not be quite correct the second or third time around either.
- Worse, each time a design phase is completed, we, the designers, will be convinced that it is error-free. A manual walk-through of the code can reveal the biggest blunders, but cannot be expected also to reveal the subtle ones. Almost by definition, we will overlook the unexpected cases that can cause errors.

The construction of a formal validation model (a prototype) and a fast and unbiased automated correctness checker is therefore indispensable for all but the simplest designs. Chapters 11 and 13 discuss how a correctness checker for PROMELA models can be constructed. In Chapter 14 the validation tool is applied to the design of this chapter, either to verify that it meets the specifications or to reveal where it is flawed. Our concern in this chapter is design and correctness. In the discussion that follows it is important to keep in mind that we are building a validation model, not an implementation. The model is an abstraction, and as such it is a design tool in itself.

A full listing of the protocol developed here can be found in Appendix F. The validation of the design with a tool called SPIN is discussed in Chapter 14.

A FILE TRANSFER PROTOCOL

The file transfer protocol we develop can be classified in a number of different ways. It is a *point-to-point* protocol, that is, it has one sender and one receiver. Protocols with more than one receiver are sometimes called *multi-point* or broadcast protocols. The file transfer protocol provides an *end-to-end* service between two users on two different machines, possibly communicating through many intermediate machines.[1]

The procedure rules for the protocol are developed as a sequence of validation models that can be checked on their correctness properties either individually or in combination. The assumption throughout this part of the design is that adequate tools are available for verifying the consistency of the intermediate validation models, and that they can be refined and adjusted with the help of those tools.

First we will make sure that the problem is completely defined. We explicitly specify the other four elements of the protocol: the service specification, the assumptions about the environment (the transmission channel), the protocol vocabulary, and the message format.

7.2 SERVICE SPECIFICATION

The protocol must implement a reliable end-to-end file transfer service. This service includes connection establishment and termination, recovery from transmission errors, and a flow control strategy to prevent the sender from overflowing the receiver. The protocol must be able to transfer ASCII text files, one at a time, with the probability of an undetected bit error being less than a modest 1 in 10^8 bits transmitted. We require that the user be able to abort a file transfer in progress, and that the protocol be able to recover from message loss.

---

1. Cf. the OSI hierarchy in Chapter 2, Section 2.6. Only layers 4 to 7 provide end-to-end service.

7.3 ASSUMPTIONS ABOUT THE CHANNEL

The protocol is to be designed for full-duplex transfer of messages over voice-grade, switched telephone lines. Having a dedicated line, rather than a network connection, we can safely ignore specific networking issues such as routing, congestion control, queueing delays, etc., and focus on concurrency control. To make it interesting, we assume that the transfers take place between New York and Los Angeles, a distance of approximately 4500 km. Given that the propagation time of an electrical signal in a cable is about 30,000 km/sec (Appendix A), the minimal time for a message to travel from sender to receiver is then approximately 0.15 seconds.

The bandwidth and average signal-to-noise ratio of a voice-grade telephone line allow us to transmit comfortably at a signaling speed of 1200 bps, using a standard modem (cf. Chapter 3 and Appendix A). We assume that transmission is character-oriented, using ASCII character encoding (Chapter 2, Section 2.5).

Most errors on telephone lines are caused by noise spikes, echoes and cross-talk. The resulting bit errors are not uniformly distributed: they come in bursts. Therefore, to describe the error characteristics, we must know some error distribution functions. In Chapter 3 (page 45) we gave several methods for predicting the probability of error-free intervals and burst durations. For the error-free intervals we are mostly interested in the average duration, not in the precise error distributions, so we will assume a simple Poisson distribution

$$Pr(EFI = n) \;=\; f \cdot e^{-f \cdot n}, \quad n \geq 0$$

where $1/f$ gives us the average duration of the error-free interval. We will use $f = 8 \cdot 10^{-6}$, which corresponds to an average error-free interval of $125{,}000$ bit transmissions (or about 100 seconds).

For the error bursts we are interested in a more accurate prediction of the probability that a given burst lasts at least n bit transmissions. We use Mandelbrot's function:

$$Pr(Burst \geq n) \;=\; \left[n^{(1-a)} - (n-1)^{(1-a)} \right] e^{-g(n-1)}, \quad 0 \leq a < 1, \quad n \geq 1$$

where a and g are parameters. The parameter values that we choose here will determine the type of error control method that will be needed. The precise values should be derived from measurements on the telephone channels used. They are, however, largely irrelevant to the discussion that follows. We will assume $a = 0.9$ and $g = 0.009$, a choice that matches our intuition about the behavior of burst errors. The average duration of a burst error that can be calculated for these parameter values is roughly 12 bit transmission times (10 msec). Figure 7.1 shows how the probability of a burst depends on its length, using the above prediction. The y-axis is logarithmic.

If we succeed in reducing the residual error rate to no more than 1 in 10^8 transmitted bits, at 1200 bps this gives an expectation of no more than one undetected error for every 23 hours of continuous operation.

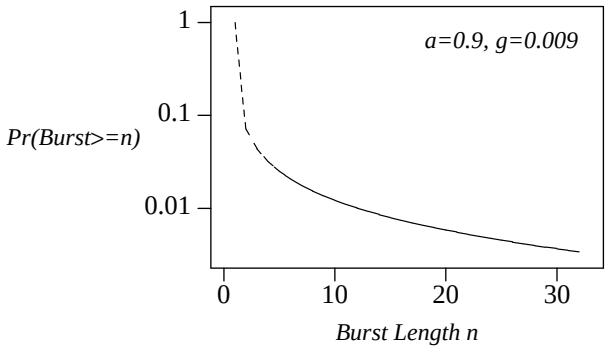

Figure 7.1 — Probability of Burst Errors

7.4 PROTOCOL VOCABULARY

Consider the protocol as a black box. To perform its function, the protocol has to communicate with its environment. It exchanges messages with the remote system via a data link, and with the local user and a local file server via internal message channels.

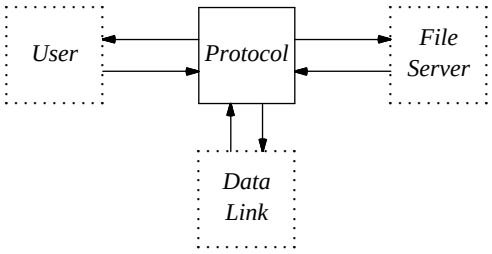

Figure 7.2 — Protocol Environment

Without going into the details of the protocol itself just yet, let us see what types of messages are needed to build it.

The black box accepts two types of messages from the local user. The first message type is used to initiate a file transfer. It can have a single parameter:

```
transfer(file_descriptor)
```

The second message type can be used by the user to interrupt a transfer in progress.

```
abort
```

The `transfer` message must trigger a message to the local file server to verify that the file to be transferred really exists, and if so, what the size of the file is. We call that message

```
open(file_descriptor)
```

If the file can be opened, its size is determined, and a connection is made with the remote system. To communicate the connection request from the local to the remote system we use the message

```
connect(size)
```

At the remote side, an incoming `connect` request again triggers a message to the file server, this time to verify that a new file of the given size can be stored. The message for that can be

```
create(size)
```

We need at least two more messages from the file server to indicate whether an `open` or a `create` request can be accepted or must be rejected:

```
accept(size)
```

and

```
reject(status)
```

In response to an `open` request, the `accept` message returns the file size to the calling process. If the request is rejected, an integer parameter can be used to carry information about the reason. The message types `accept` and `reject` can also be used to inform the local user about the success or failure of an outgoing file transfer. And similarly, they can be used by the local system to inform a remote system whether an incoming file transfer is accepted.

To transfer a file, four things have to happen in a specific order:

- Establishment of a connection with the local file server
- Establishment of a connection with the remote system
- Transfer of data
- Orderly termination of the connection

Each protocol phase has its own vocabulary of messages and its own rules for interpreting them. Each phase may again have to be sub-divided into still smaller steps. The second phase, for instance, will consist of two steps: (1) an initialization of the flow control protocol, and (2) a handshake with the remote system using the `connect` and `accept` or `reject` message. The synchronization of the local and the remote flow control layer protocols is necessary to guarantee that they agree on the initial sequence numbers to be used. We can use the message

```
sync
```

and its acknowledgment

```
sync_ack
```

for this purpose.

In the third protocol phase we need messages for retrieving the data from the file server and transmitting them to the remote system. For example, we can use a message

```
data(cnt, ptr)
```

to transfer `cnt` bytes of information, available in a data buffer that is identified by the second parameter. This, of course, is not necessarily the way in which interactions with the file server have to be implemented in a final design. For now, however, it suffices that it accurately models the essentials of these interactions.

Another message,

```
eof
```

can be used by the file server to signify the end of a file. The correct completion of a

file transfer, with the `eof` message, can be confirmed by the remote system with a single message

`close`

So far, the only assumption we have made about the file server is that it recognizes six messages: `open`, `create`, `accept`, `reject`, `data`, and `eof`. To avoid synchronization problems between the local system and the local file server, we assume that the above six messages are exchanged by rendezvous communication (i.e., they are equivalent to local procedure calls). This guarantees that, for instance, unused `data` messages from or to the file server do not accumulate. After the successful completion of both a local `open` and a remote `create` request, the `data` messages are used to transfer data from local file server to the remote file server, using the protocol to be developed.

We need one extra message to implement a simple flow control discipline that acknowledges correctly received data:

`ack`

The complete protocol vocabulary then consists of thirteen distinct message types. Nine of these messages can be exchanged by the two remote machines: `accept`, `ack`, `close`, `connect`, `data`, `eof`, `reject`, `sync_ack`, and `sync`. The other four, `abort`, `create`, `open`, and `transfer`, are internal messages only.

7.5 MESSAGE FORMAT

It should now be decided what the right format for the above messages is. The messages minimally require a type field and an optional data field. To implement a flow control discipline, the messages sent to the remote system must also carry sequence numbers, and to implement error control they must carry a checksum field. Since the transmission channel is byte oriented, we can format each message as a sequence of bytes. Clearly, we would like these sequences to be as short as possible. Let us see how we can calculate the minimal size required, knowing only what we know so far about the protocol and the transmission channel to be used.

At the highest level, a message can be encoded into a series of bytes, indicating its type and the value of its parameters. Before the message is sent, the flow control appends a sequence number, and the error control appends a checksum field. At the lowest level, just before the message is placed onto the transmission line, a line driver appends the message delimiters, to enable the remote system to recognize where a message starts and stops. The ASCII 8-bit patterns `STX` (start of text) and `ETX` (end of text) can be used for this purpose. With byte stuffing (Chapter 2), misinterpretation message delimiters that are part of the data itself can be avoided.

The remote receiver strips the `STX` and `ETX` characters and removes the stuffed characters. The remote error control strips and interprets the checksum, and the remote flow control strips and interprets the sequence number. If all is well, the message finally arrives at its peer as the original series of bytes again.

The best position of the various message fields in the byte sequence depends to some

extent on the encoding of the protocol routines at the lowest level (the physical layer) in the hierarchy. Placing the checksum field at the end of a message has the advantage that the sender can compute it on the fly while transmitting the body of the message. There can also be a small difference in performance depending on where the message-type field is placed. It can either be placed at the front of the message, in a fixed place behind the `STX` symbol, or at the end, in a fixed place before the `ETX` symbol. Since we have variable length messages, the position of the `ETX` symbol is less predictable than that of the `STX` symbol. Placing the type field at the start of the message can therefore make it easier to parse an incoming message.

We thus arrive at the message format shown in Figure 7.3, where the *Data* field is absent in control messages.

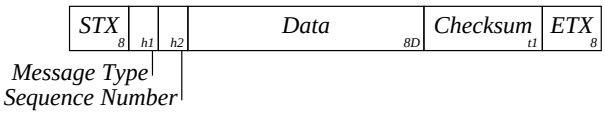

Figure 7.3 — Message Format

In Figure 7.3, a symbol representing the length of each field in bits is indicated in the lower-righ corner. `D` bytes of information in the data field corresponds to a field width of `8D` bits. How are the numbers `h1`, `h2`, `D`, and `t1` determined?

THE MESSAGE TYPE FIELD

This is the easiest of the four numbers to calculate. We have 13 different types of message. Since $2^3 < 13 \leq 2^4$, 4 bits for the type field suffice.

$$h_1 = 4$$

THE SEQUENCE NUMBER FIELD

Now, let us derive an appropriate width for the sequence number field h_2. The message propagation time is 0.15 seconds, long enough to transmit 180 bits. If a message would be bounced back by the receiver without processing delays, the sender could transmit 360 bits before receiving the return message. We would like to design the protocol in such a way that the sender can completely saturate the channel whenever the receiver consumes the data as fast as the sender produces data. We must therefore make certain that the sender will not have to wait idly for acknowledgments when it could be sending messages. This means that the sender should be able to be at least 360 bits ahead of the receiver under normal circumstances. The number of messages that this corresponds to depends on the number of bytes in the data field and in the header and trailer.

If we use a selective repeat continuous ARQ method (Chapter 4), the most flexible flow control method we have discussed, the maximum number of outstanding messages with a sequence number field of h_2 bits is 2^{h_2-1}. With a one-bit sequence number only one message can be outstanding at a time, forcing the sender to remain

idle for at least 0.3 seconds, waiting for the acknowledgment on that message (twice the message propagation time). With a two-bit sequence number, two messages may be outstanding, meaning that the sender can transmit one message while awaiting the acknowledgment for another. If that message is at least 360 bits long, no time is lost. Below we will see that it is indeed advantageous to use a data field longer than 360 bits. So, we have

$$h_2 \geq 2$$

Assuming again that the receiver is at least as fast as the sender, we could organize the flow control as follows. All messages within the current window are sent before the acknowledgment for the oldest outstanding message is checked. If no acknowledgment is received by that time, the oldest message is retransmitted. If an acknowledgment is received, the window slides up one notch and a new message can be transmitted. The sender is never idle, and the channel is saturated.

Note carefully that if the receiver is slower than the sender it is prudent not to try to saturate the channel: the receiver needs time to catch up with the sender. For this case we can include a timer. If no acknowledgment for the oldest outstanding frame is received by the time the sender checks for one, the sender now waits at least an additional timeout period for the acknowledgment to arrive. The appropriate value for the timeout count can be estimated, or it can be adjusted dynamically with a *rate control* method. We shall restrict ourselves to the optimal case, with a fully saturated channel.

THE DATA FIELD

The length of the data field is expressed in the byte count D and is variable per data message. It is not to our advantage to make a message too long, since the expectation that a message contains bit errors trivially increases with its length. On the other hand, if we make the messages too small, the overhead[2] of the header and trailer becomes too large. The average duration of an error-free interval was estimated at approximately $125,000$ bit transmissions, so we certainly should not make a message longer than that. Somewhere between these 125 Kbits and the 360 bits we derived earlier there must be an optimal length for the data field. We can approximate this optimum as follows.

OPTIMAL DATA SIZE

Let t be the length of the data message overhead in bits (header plus trailer) and d the length of the data field, also in bits: $d = 8D$. Further, let a be the length of an `ack` control message, p_d the probability of a data message being distorted or lost, and p_a the probability of the same error for an acknowledgment.

Let us first consider the case where $p_d = p_a = 0$. The transmission of one message requires one data message and one acknowledgment message, a total of $d + t + a$ bits.

2. See also ‘‘code rates’’ defined in Chapter 3, Section 3.5.

The overhead is $t + a$, and the protocol efficiency is

$$E = \frac{d}{d + t + a}$$

So in the absence of errors, it is best to chose d as large as possible.

Now consider the case where p_d and p_a are non-zero. The probability that neither the data message nor its acknowledgment is hit by a transmission error is $(1 - p_d)(1 - p_a)$. Therefore, the probability that the message must be retransmitted is

$$p_r = 1 - (1 - p_d)(1 - p_a)$$

The probability that it takes i subsequent transmissions to get the data message across, $i - 1$ retransmissions and one successful transmission, is

$$p_i = (1 - p_r)\ p_r^{i-1}$$

The expected number of transmissions per message R is then given by

$$\begin{aligned} R &= \sum_{i=1}^{\infty} i p_i \\ &= \sum_{i=1}^{\infty} i(1 - p_r) p_r^{i-1} \\ &= (1 - p_r) \sum_{i=1}^{\infty} i p_r^{i-1} \\ &= (1 - p_r) \sum_{j=0}^{\infty} \sum_{i=j}^{\infty} p_r^i \\ &= (1 - p_r) \sum_{j=0}^{\infty} \frac{p_r^j}{1 - p_r} \\ &= \frac{1}{1 - p_r} \end{aligned}$$

The relative efficiency E, in the presence of errors, is then

$$E = \frac{d}{R(d + t + a)}$$

The optimal value for d can be found by setting the derivative of E with respect to d to zero: $\delta E / \delta d = 0$. Alternatively, we can simply fill in the known or approximate values for R, t, and a, and plot E as a function of d.

With the values we derived earlier, and counting 8 bits each for the `STX` and `ETX` message delimiter, we have

$$\begin{aligned} a &= h_1 + h_2 + t_1 + 16 \\ &= 4 + 2 + 16 + 16 = 38 \end{aligned}$$

and trivially

$$t = a = 38$$

We earlier assumed that an error-free interval would last an average of 125 Kbits. This corresponds to $(125 \cdot 10^3)/(d+t)$ messages or $(125 \cdot 10^3)/a$ acknowledgments. A first-order approximation can then be obtained by taking

$$p_d = (d+38)/(125 \cdot 10^3)$$

and

$$p_a = 38/(125 \cdot 10^3) = 3.04\ 10^{-4}$$

We now have

$$R = \frac{1}{1-p_r}$$

$$= \frac{1}{(1-p_d)(1-p_a)}$$

Substituting in the last expression for E gives:

$$E = \frac{d \cdot (1-p_d)(1-p_a)}{d+t+a}$$

$$= \frac{d \cdot (1-(d+38)/(125 \cdot 10^3))(1-38/(125 \cdot 10^3))}{d+76}$$

In Figure 7.4 the protocol efficiency E is plotted as a function of the length of the message data field d.

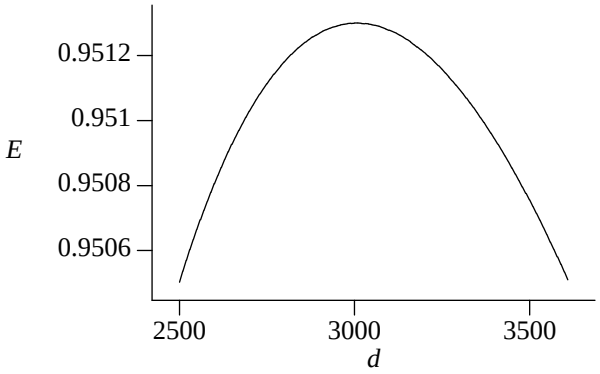

Figure 7.4 — Protocol Efficiency versus Data Size in Bits

There is clearly an optimum value for d. The maximum efficiency of 95.13% is achieved for a data size of 3004 bits, or 376 bytes. So we have now derived the third value D for Figure 7.3.

In reality, of course, the probability of lost and distorted data should be derived from the error distribution functions. We should have solved the more difficult equations

$$p_d = \sum_{i=0}^{d+38} Pr(EFI = i) \quad p_a = \sum_{i=0}^{38} Pr(EFI = i)$$

To see how large the mistake is we made, we set $d = 3004$ and calculate

$$p_d = \sum_{i=0}^{3004+38} Pr(EFI = i) = 240.42\ 10^{-4}$$

and

$$p_a = \sum_{i=0}^{38} Pr(EFI = i) = 3.039\ 10^{-4}$$

The earlier approximations give for the same value of d

$$p_d = (3004+38)/(125 \cdot 10^{3)} = 243.36\ 10^{-4}$$

and

$$p_a = 38/(125 \cdot 10^{3)} = 3.04\ 10^{-4}$$

Our first estimate is within 1.2 percent of the recalculated values. Since we have no reason to trust the predictive value of the error distribution function with that degree of accuracy, we settle for the first estimate.

THE CHECKSUM FIELD

All that remains is to derive the value for t_1, the width of the checksum field. The channel has deletion and distortion errors, but is not expected to produce insertions or message reorderings. The error rate is low enough that we do not need an error correcting code. For a reasonable message length, well below 125 Kbits, most messages get through without transmission errors. We must, however, be able to correct for the characteristic errors of the channel: burst errors. This makes a cyclic redundancy check a good choice. The average duration of a burst error was assumed to be 10 msec, affecting a sequence of 12 bits. The degree of the generator polynomial therefore at least has to be larger than 12 to catch these errors.

The target residual bit error rate is 10^{-8}. To be able to check if we can comply with this requirement by choosing the 16-bit CRC-CCITT checksum polynomial, we have to look at the distribution of burst-error durations. We know that the CRC-CCITT checksum catches all single and double bit errors, all odd numbers of bit errors, all burst errors up to 16 bits long, 99.997% of burst errors of 17 bits, and 99.998% of all burst errors longer than 17 bits.

We assumed that burst errors occur on the average once every 100 seconds. A burst of roughly 12 bits long then corresponds to a long-term average bit error rate of 10^{-4}. Of the bursts longer than 16 bits, two or three out of every 10^5 bursts will go undetected. If *every* burst was longer than 16 bits, this would mean that two or three

burst errors per 100×10^5 seconds would get through, corresponding to a long-term average bit error rate of roughly 10^{-6} (there are more than ten bits in every burst). We can therefore only realize a target residual bit-error rate of 10^{-8} if burst errors longer than 16 bits occur less than once out of every 10^2 bursts, or fewer than once every 10,000 seconds. Using the probability distribution function, we find (cf. Figure 7.1):

$$Pr(Burst \geq 17) \quad = \quad 0.009.e^{-0.009 \cdot 17} \quad = \quad 0.007$$

The result is within range of the target. We can choose the 16-bit checksum, and have our last value:

$$t_1 \quad = \quad 16$$

For the control messages it seems like overkill to include even a 16-bit checksum for a message that carries only two small numbers. Note, however, that a burst error can wipe out the complete message, so with the same redundancy an error correcting code could not perform better.

As an aside, the message format, with all the field widths we have now derived, can be defined in C as follows.

```
struct {
        unsigned type  : 4;
        unsigned seqno : 2;
        unsigned char data[376];
        unsigned char checksum[2];
} message;
```

We used bit-fields for the fields in the message header and unsigned characters (8 bits wide) for the data field and the checksum. The message delimiters STX and ETX were omitted. This is, however, not necessarily the way in which a C compiler would arrange for the bits to be stored in memory. Some padding may occur to align bit-fields with word or byte boundaries.

EFFECTS OF ROUNDING

In Figure 7.4 we see that the protocol efficiency is not very sensitive to variations in the data size near the optimum. We are also stuck with the peculiar values for h_1 and h_2 that we calculated earlier. Multiples of 8 bits would be more convenient for the receiver to process. Let us see how badly the efficiency would be affected if we used the nearest multiple of 8 for all field widths, as shown in Table 7.1.

Table 7.1 — Rounding

Symbol	Old	New
h_1	4	8
h_2	2	8
t_1	16	16

The recalculated values for E as a function of d for these new values are indicated by

the lower curve in Figure 7.5. The effect of adding 10 bits of overhead is a reduction of E by less than one percent, which should be considered insignificant compared to the errors introduced by earlier approximations. The new optimum is reached for a data length of 3370 bits, or about 422 bytes (46 bytes more than before).

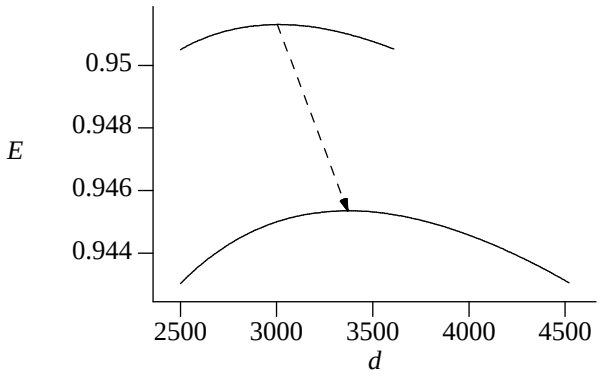

Figure 7.5 — Degradation of Protocol Efficiency by Rounding

After rounding, the message format can be written without bit-fields

```
struct {
        unsigned char type;
        unsigned char seqno;
        unsigned char data[422];
        unsigned char checksum[2];
} message;
```

7.6 PROCEDURE RULES

We are now ready to start with the description of the procedure rules. In this part of the design, few things can be calculated or measured. Yet the rules have to be complete and consistent. Before committing ourselves directly to an implementation, we would like to be able to design and debug the rules in an intermediate form, for instance as a validation model. PROMELA was designed for precisely this purpose. We first look at some of the abstractions we have to make in the modeling of the messages. Then we look at the layering of the protocol and derive a rough global structure. Finally, from the highest layer down to the lowest we refine each layer and combine them into the final design. The correctness requirements for each layer are formalized using the notation developed in Chapter 6.

ABSTRACTION

The precise encoding for the messages, derived in the previous sections, contains too much information for the problem we are now facing. Manipulating messages at this level of detail would needlessly complicate the design, description, and validation of the procedure rules. To derive the procedure rules we build a model of the

communication system in which we consider only the semantics of the protocol, not its precise syntax. To the model, for instance, the contents of transferred files are irrelevant. The variable length `Data` field from Figure 7.3, therefore, need not be represented, nor (as we will argue below) the checksum field. We use a message template of just two fields in the validation model.

```
fld1,fld2
```

The first field represents the generic message type, e.g., `data`, and the second field carries a parameter, such as a sequence number. Both fields can be of PROMELA type `byte`.

What we are designing from this point on is a validation model, not an implementation. If we do our job right though, it should be simple to remove the abstractions from the model, refine it where necessary, and derive an implementation.

LAYERS

To structure the design, we divide it into several layers. The user interacts with a presentation layer protocol. Below that we place a session control layer, and below that a data link layer that enforces a general flow control discipline. The data link is the physical data line, equipped with modems, for encoding binary data into analog signals, and providing error detection on every message transmitted using CRC-CCITT checksumming. The data link we work with from this point on, therefore, can lose but not distort messages.

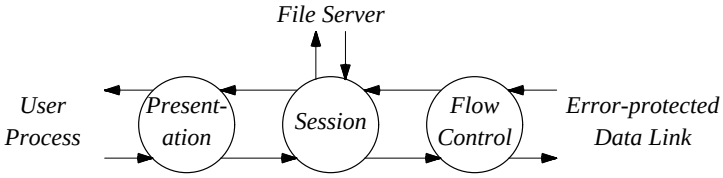

Figure 7.6 — Protocol Hierarchy

Each layer in this hierarchy is managed by one or more PROMELA processes, as indicated with the protocol ‘‘pipeline’’ in Figure 7.6.

PROTOCOL ENVIRONMENT

Our first job in building a validation model is to make explicit all relevant assumptions about the behavior of the environment, as illustrated in Figures 7.2 and 7.6. The environment consists of three entities:

- User process
- File server
- Data link

The minimal assumptions we must make about the behavior of each of these three is formalized in PROMELA code.

First consider the user level protocol. There can be two user processes: one on each

end of the data link. The users can submit a transfer request at any time by passing a file descriptor to the presentation layer of the protocol. At any time after the transfer request, the originating user may also decide to abort a transfer. We assume that the user then waits for a response from the lower protocol layers, signaling either the successful or unsuccessful completion of the transfer. We can model these assumptions with the following PROMELA process.

```
proctype userprc(bit n)
{       use_to_pres[n]!transfer;
        if
        :: pres_to_use[n]?accept -> goto Done
        :: pres_to_use[n]?reject -> goto Done
        :: use_to_pres[n]!abort  -> goto Aborted
        fi;
Aborted:
        if
        :: pres_to_use[n]?accept -> goto Done
        :: pres_to_use[n]?reject -> goto Done
        fi;
Done:
        skip
}
```

The binary argument `n` identifies the user and the channels that it accesses. The message `transfer` would ordinarily carry a parameter that points to the file to be transferred (e.g., a file-descriptor). To a validation, however, the value of that parameter is irrelevant, and it is therefore not present in the model. Clearly, the correctness of the file transfer protocol should not depend on the particular file-descriptors used.

The message channels we use in the model can be defined globally. Every arrow in Figure 7.6 corresponds to two such channels, one for each side of the protocol. Using a simple naming scheme to indicate which layers each channel connects, we can define the following types of channels. There are two users, and `QSZ` is a queue size of at least one (cf. Figure 7.6).

```
chan use_to_pres[2] = [QSZ] of { byte };
chan pres_to_use[2] = [QSZ] of { byte };
chan pres_to_ses[2] = [QSZ] of { byte };
chan ses_to_pres[2] = [QSZ] of { byte, byte };
chan ses_to_flow[2] = [QSZ] of { byte, byte };
chan flow_to_ses[2] = [QSZ] of { byte, byte };
chan dll_to_flow[2] = [QSZ] of { byte, byte };
chan flow_to_dll[2] = [QSZ] of { byte, byte };
```

The channels for the synchronous communication between the session layer protocol and the file server are defined with zero buffer capacity, as follows:

```
chan ses_to_fsrv[2] = [0] of { byte };
chan fsrv_to_ses[2] = [0] of { byte };
```

This brings the total to ten different types of message channels, with one copy being instantiated for each side of the connection. The channels used for the higher protocol layers can be defined with a simpler message format than the lower layers. The

sequence number field, for instance, is used only by the flow control layer.

Next we must formalize our assumptions about the behavior of the file server. Again, no design decisions are made yet. The aim is merely to make explicit what must minimally be assumed about the external behavior of the file server process. An incoming file transfer begins with a `create` message. The file server responds with either a `reject` or an `accept` message. As far as the validation model is concerned, either choice is equally likely, so it can be modeled as a nondeterministic one. If the request is accepted, zero or more `data` messages follow, and the file server falls back into its initial state upon the reception of the final `eof` or, in case of an abort, the `close` message. In first approximation, disregarding what we said earlier about abstraction, we may try to describe this behavior in a PROMELA validation model as follows.

```
proctype fserver(bit n)
{       int fd, size, ptr, cnt;

        do
        :: ses_to_fsrv[n]?create(size) ->       /* incoming */
                if      /* nondeterministic choice */
                :: fsrv_to_ses[n]!reject
                :: fsrv_to_ses[n]!accept ->
                        do
                        :: ses_to_fsrv[n]?data(cnt,ptr)
                        :: ses_to_fsrv[n]?eof -> break
        /* abort */     :: ses_to_fsrv[n]?close -> break
                        od
                fi
        :: ses_to_fsrv[n]?open(fd) ->   /* outgoing */
                ...
        od
}
```

But we are on the wrong track with this model. Notice that local variables and message parameters `fd`, `size`, `cnt`, and `ptr` really provide unwanted detail in the model. We are designing the procedure rules for the interaction of the file server with the session layer protocol. We want to specify how these two modules interact, i.e., the types of messages that they will exchange and the inherent expectations about the order in which the different types of messages will arrive and will be sent. Important to specify at this level of abstraction is *when* data can be passed, not *which* data will be passed. The size and contents of data transferred are therefore still irrelevant at this level of modeling (cf. Chapter 14 about the formal generalization of models).

A better way to model the relevant behavior of the file server for incoming data is

```
proctype fserver(bit n)
{       do
        :: ses_to_fsrv[n]?create ->     /* incoming */
```

```
                if
                :: fsrv_to_ses[n]!reject
                :: fsrv_to_ses[n]!accept ->
                        do
                        :: ses_to_fsrv[n]?data
                        :: ses_to_fsrv[n]?eof -> break
                        :: ses_to_fsrv[n]?close -> break
                        od
                fi
        :: ses_to_fsrv[n]?open ->       /* outgoing */
                ...
        od
}
```

The model for outgoing data is similar. After the server receives an `open` message, it may respond with either an `accept` or a `reject` message. If the request is accepted, a series of `data` messages is transferred, followed by a single `eof` message. The file transfer can be aborted again by a `close` message from the session layer to the file server.

```
        :: ses_to_fsrv[n]?open ->               /* outgoing */
                if
                :: fsrv_to_ses[n]!reject
                :: fsrv_to_ses[n]!accept ->
                        do
                        :: fsrv_to_ses[n]!data
                        :: fsrv_to_ses[n]!eof -> break
                        :: ses_to_fsrv[n]?close -> break
                        od
                fi
```

The last piece of the environment is the data link. Again, we must make explicit all our assumptions about its behavior, in so far as it relates to the protocol we are designing.

The data link is assumed to be protected with an error detection protocol. The details of the checksum calculation can be found in Chapter 3, Section 3.7, but for this part of the design problem these details are irrelevant. The checksum calculation is a computation, and not an interaction pattern. It would be folly to try to model it in detail in PROMELA as if it were a procedure rule.

All we are interested in here is the external behavior of the data link. The data link, then, can arbitrarily omit messages from the sequences that it passes, using some hidden oracle to decide the fate of each message. The choice can be modeled as a nondeterministic one.

```
proctype data_link()
{   byte type, seq;
```

```
    do
    :: flow_to_dll[0]?type,seq ->
        if
        :: dll_to_flow[1]!type,seq
        :: skip /* lose */
        fi
    :: flow_to_dll[1]?type,seq ->
        if
        :: dll_to_flow[0]!type,seq
        :: skip /* lose */
        fi
    od
}
```

We have now completed the formalization of all assumptions about the environment in which the protocol must be used. We are now ready to design the three core protocol layers: the presentation, the session layer, and the flow control layer.

7.6.1 PRESENTATION LAYER

The presentation layer provides the interface to the user. Its job is to take care of the details of the file transfer, providing, for instance, for the resubmission of the request for file transfer when a non-fatal error occurs. We can anticipate five different reasons for an outgoing file transfer request to fail.

1. The local system is busy serving an incoming file transfer.
2. The local file server rejects the request, for instance because the file to be transferred does not exist.
3. The remote file server rejects the request, for instance because it cannot allocate sufficient space.
4. There is a collision between an incoming and an outgoing file transfer request.
5. The file transfer was aborted by the user.

Two of these five possible causes of a failure are transient (1 and 4) and may disappear if the transfer request is repeated. The presentation layer can further try to protect itself against repeated abort requests from the user processes by filtering out any duplicates.

To get started on a design for the presentation layer we can draw a tentative state diagram. The presentation layer, then, can be given two main states: an `IDLE` state and a busy, or `TRANSFER` state. The process moves from `IDLE` to `TRANSFER` upon the reception of the user's `transfer` request. It returns to the `IDLE` state upon the successful completion of the transfer or the detection of a fatal error. This initial outline is shown in Figure 7.7.

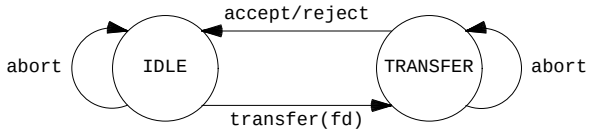

Figure 7.7 — Partial State Transition Diagram: Presentation Layer

Using Figure 7.7 as a guideline we can fill in the relevant details in a validation model as follows.

```
#define FATAL           1       /* failure types */
#define NON_FATAL       2       /* repeatable    */
#define COMPLETE        3       /* success       */

proctype present(bit n)
{       byte status, uabort;

IDLE:
        do
        :: use_to_pres[n]?transfer ->
                uabort = 0;
                goto TRANSFER
        :: use_to_pres[n]?abort ->
                skip    /* ignore */
        od;
TRANSFER:
        pres_to_ses[n]!transfer;
        do
        :: use_to_pres[n]?abort ->
                if
                :: (!uabort) ->
                        uabort = 1;
                        pres_to_ses[n]!abort
                :: (uabort) ->
                        skip
                fi
        :: ses_to_pres[n]?accept ->
                goto DONE;
        :: ses_to_pres[n]?reject(status) ->
                if
                :: (status == FATAL || uabort) ->
                        goto FAIL
                :: (status == NON_FATAL && !uabort) ->
                        goto TRANSFER
                fi
        od;
DONE:
        pres_to_use[n]!accept;
        goto IDLE;
```

```
FAIL:
        pres_to_use[n]!reject;
        goto IDLE
}
```

The file transfer request is repeated until it succeeds or until it triggers a fatal error. The main assumption that the presentation layer makes about the session layer protocol is that it will eventually respond to a transfer request with either an `accept` or a `reject` message. It also assumes that the session layer can accept an `abort` message at any time.

Correctness Requirements: As a correctness requirement for the presentation layer we will identify its valid end-states. There is one valid end-state only, the `IDLE` state. By adding the prefix ''end,'' i.e. replacing the name `IDLE` with `endIDLE`, we can specify the requirement that the presentation layer must be in the given state when a protocol transfer terminates. Provided that our assumptions about the session layer and the user layer are true, it is not too hard to show that this requirement is satisfied. The presentation layer can be blocked only in a small number of unexecutable statements. The assumptions about the user and session layers guarantee that none of those statements can remain unexecutable forever. But an informal argument, such as the one above, is not a proof. In Chapter 14 we use an automated prover to show that the assumptions we have made and the conclusions we have drawn from them are not just convincing but also correct.

7.6.2 SESSION LAYER

Earlier we decided that the protocol would have four phases:

- Initialization of the file server
- Connection establishment
- Data transfer
- Call completion

The messages in the protocol vocabulary that we looked at earlier are, of course, not unrelated; they imply a certain ordering. A transfer request should precede a connection confirmation, just as connection establishment should precede data transfer. We can make some of these implied relations explicit with the outline shown in Figure 7.8.

The transitions are labeled with numbers that refer to some, though not all, of the messages that are exchanged. Incoming messages are prefixed with a question mark, outgoing messages with an exclamation point. The dashed arrows indicate the expected sequence of events for a successful outgoing file transfer; the dotted lines indicate a normal incoming file transfer. Writing down the state transition diagram has forced us to make some further design choices. In the call establishment phase, for instance, a connection request may come from the remote system directly after the processing of a local connection request has started. In Figure 7.8 we have chosen to reject both requests when such a call collision occurs. The call completion phase is entered when a file has been successfully transferred, and the `eof` message has been sent. Alternatively, the transfer can be interrupted when the user sends the `abort`

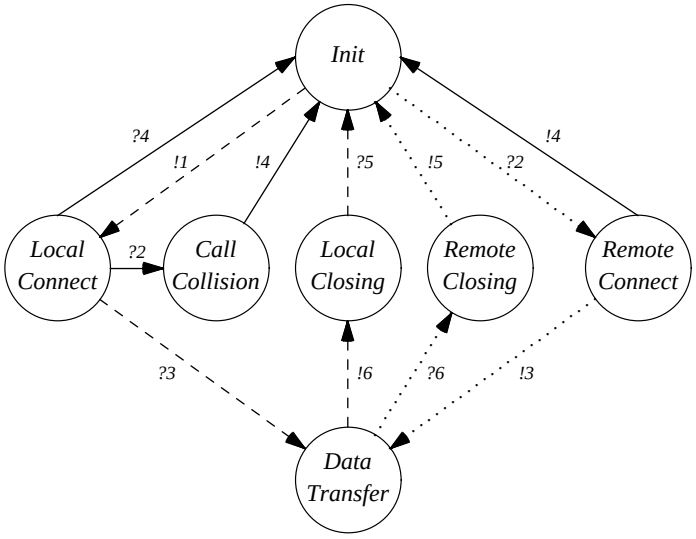

Figure 7.8 — Partial State Transition Diagram: Session Layer
1: transfer, 2: connect, 3: accept, 4: reject, 5: closed, 6: eof

message. In the following paragraphs we consider each protocol phase separately and fill in the details.

CALL ESTABLISHMENT PHASE

The session layer sits between the presentation and the flow control layer. After the local `transfer` message from the presentation layer arrives, the session layer must perform a number of crucial tasks. It must try to open the file for reading on the local system, it must create a file for writing on the remote system, and in between it must establish a connection with the remote system. Any of these three tasks may fail. The local file server takes care of all file access.

This leads to the following descriptions for the call establishment phase. First, there are only two messages that can trigger a file transfer: the local `transfer` message and the remote `connect` message. Everything else should be ignored by the session protocol. Instead of explicitly specifying all non-valid messages, we can make the session layer receive a message of arbitrary type and check only for a match with the expected message, which in state `IDLE` is either a `transfer` message coming from the upper protocol layer or a `connect` message from the lower layer.

```
proctype session(bit n)
{       bit     toggle;
        byte    type, status;

IDLE:
```

```
        do
        :: pres_to_ses[n]?type ->
                if
                :: (type == transfer) ->
                        goto DATA_OUT
                :: (type != transfer)
                        /* ignore */
                fi
        :: flow_to_ses[n]?type ->
                if
                :: (type == connect) ->
                        goto DATA_IN
                :: (type != connect)
                        /* ignore */
                fi
        od;
DATA_OUT:
        ...
DATA_IN:
        ...
}
```

It seems best to separate the data transfer phases for incoming and outgoing files. The easier case is the preparation for an incoming file. There is only one interaction with the local file server.

```
DATA_IN:                /* prepare local file server */
        ses_to_fsrv[n]!create;
        do
        :: fsrv_to_ses[n]?reject ->
                ses_to_flow[n]!reject;
                goto IDLE
        :: fsrv_to_ses[n]?accept ->
                ses_to_flow[n]!accept;
                break
        od;
        ... incoming data transfer ...
        ... close connection etc.  ...
```

An outgoing transfer is done in three steps, each of which can fail:

- Handshake on a local `open` request with the file server
- Initialization of the flow control layer
- Handshake with the remote system on a `connect` request

The first step can be specified as follows.

```
DATA_OUT:               /* 1. prepare local file */
        ses_to_fsrv[n]!open;
        if
        :: fsrv_to_ses[n]?reject ->
                ses_to_pres[n]!reject(FATAL);
                goto IDLE
```

```
:: fsrv_to_ses[n]?accept ->
        skip    /* proceed */
fi;
```

The second step is harder. We must make sure that we cannot accidentally accept an old `sync_ack` message from a previous initialization attempt. We use a one-bit sequence number to solve that problem (see Exercise 7-12.)

```
                /* 2. initialize flow control */
ses_to_flow[n]!sync,toggle;
do
:: flow_to_ses[n]?sync_ack,type ->
        if
        :: (type != toggle) /* ignore */
        :: (type == toggle) -> break
        fi
:: timeout ->   /* failed */
        ses_to_fsrv[n]!close;
        ses_to_pres[n]!reject(FATAL);
        goto IDLE
od;
toggle = 1 - toggle;
```

In the third and last step, we must consider the possibility of a call collision.

```
                /* 3. prepare remote file */
ses_to_flow[n]!connect;
if
:: flow_to_ses[n]?accept ->
        skip    /* success */
:: flow_to_ses[n]?reject ->
        ses_to_fsrv[n]!close;
        ses_to_pres[n]!reject(FATAL);
        goto IDLE
:: flow_to_ses[n]?connect ->
        ses_to_fsrv[n]!close;
        ses_to_pres[n]!reject(NON_FATAL);
        goto IDLE
:: timeout ->   /* got disconnected? */
        ses_to_fsrv[n]!close;
        ses_to_pres[n]!reject(FATAL);
        goto IDLE
fi;
... outgoing data transfer...
... close connection etc. ...
```

When a call collision is detected both parties close their files and return to the initial state, leaving it up to the presentation layers to make another attempt to transfer their files. Prudence dictates that we include a `timeout` clause in every state where we wait for events that can only be supplied by the remote system. In case all communication is lost, e.g., on carrier loss, it gives us a way back to the initial state.

DATA TRANSFER PHASES

The design of the data transfer phases is relatively straightforward. For outgoing transfers we obtain data from the file server and transfer them to the flow control layer. The only things that can complicate the design are the messages that can interrupt the transfer: a `timeout` or an `abort` message from the user.

```
do                      /* outgoing data */
:: fsrv_to_ses[n]?data ->
        ses_to_flow[n]!data
:: fsrv_to_ses[n]?eof ->
        ses_to_flow[n]!eof;
        status = COMPLETE;
        break   /* goto call termination */
:: pres_to_ses[n]?abort ->      /* user aborts */
        ses_to_fsrv[n]!close;
        ses_to_flow[n]!close;
        status = FATAL;
        break   /* goto call termination */
od;
```

For incoming file transfers, we obtain data messages from the flow control layer and transfer them to the file server. Only the remote user can abort an incoming transfer. When the remote user aborts, the local system receives a `close` message from the remote system (see above). This leads to the following PROMELA model.

```
do                      /* incoming data */
:: flow_to_ses[n]?data ->
        ses_to_fsrv[n]!data
:: flow_to_ses[n]?eof ->
        ses_to_fsrv[n]!eof;
        break
:: pres_to_ses[n]?transfer -> /* sorry, busy */
        ses_to_pres[n]!reject(NON_FATAL)
:: flow_to_ses[n]?close ->      /* remote user aborts */
        ses_to_fsrv[n]!close;
        break
:: timeout ->   /* got disconnected? */
        ses_to_fsrv[n]!close;
        goto IDLE
od;
```

All the necessary character stuffing and byte encoding operations (Chapter 2, Section 2.5) are again irrelevant to the validation model. We can safely assume that they happen elsewhere, e.g., in the modem that connects us to the physical line.

CALL TERMINATION PHASE

We complete the high-level design of the data transfer phase by adding the processing of call termination messages. The call termination phase can only be entered from the data transfer phase, as shown above. First let us consider the normal termination of an outgoing file transfer session.

```
/* close connection, outgoing transfer */
do
:: pres_to_ses[n]?abort /* too late, ignored */
:: flow_to_ses[n]?close ->
        if
        :: (status == COMPLETE) ->
                ses_to_pres[n]!accept
        :: (status != COMPLETE) ->
                ses_to_pres[n]!reject(status)
        fi;
        break
:: timeout ->   /* disconnected? */
        ses_to_pres[n]!reject(FATAL);
        break
od;
goto IDLE
```

The code for responding to the termination of an incoming file transfer is simpler:

```
/* close connection, incoming transfer */
ses_to_flow[n]!close;   /* confirm it */
goto IDLE
```

Correctness Requirements: The main correctness requirement for the session control layer protocol is that it satisfies the assumptions made by the presentation layer protocol: it always responds to a `transfer` message, within a finite amount of time, with either an `accept` or a `reject` message. Similarly, a remote `connect` message should be followed by an `accept` or a `reject` message to the remote presentation layer, within a finite amount of time. We can formalize both these requirements in a temporal claim by defining behavior that is required to be absent. A first attempt is shown below.

```
never {
        do
        :: !pres_to_ses[n]?[transfer]
        && !flow_to_ses[n]?[connect]
        :: pres_to_ses[n]?[transfer] ->
                goto accept0
        :: flow_to_ses[n]?[connect] ->
                goto accept1
        od;
accept0:
        do
        :: !ses_to_pres[n]?[accept]
        && !ses_to_pres[n]?[reject]
        od;
accept1:
        do
        :: !ses_to_pres[1-n]?[accept]
        && !ses_to_pres[1-n]?[reject]
        od
}
```

where `n` is either zero or one. An automated validation of the session layer protocol

based on this claim is discussed in Chapter 14. We can also identify the `IDLE` state in the session layer as a valid end-state, again by simply replacing the name with `endIDLE`.

To complete the design, we now provide the data link layer, which comes in two parts: one layer implementing a flow control discipline and one providing error control.

7.6.3 FLOW CONTROL LAYER

We can model a full sliding window protocol, as discussed in Chapter 4, by directly encoding Figures 4.11 and 4.12. To make things interesting, we will restrict ourselves to an encoding with a single process instead of five. In this case we cannot get away with an abstraction for parameters such as the window size and the range of sequence numbers used, without losing essential information about the operation of this protocol layer. All data in PROMELA is initialized to zero by default. We discuss the model step by step.

```
#define true    1               /* for convenience */
#define false   0

#define M       4               /* range sequence numbers   */
#define W       2               /* window size: M/2         */
```

As we saw in Chapter 4, and will prove in Chapter 11, the maximum number of outstanding messages in a protocol of this type is equal to half the range of the sequence numbers.

```
proctype fc(bit n)
{       bool    busy[M];        /* outstanding messages    */
        byte    q;              /* seq# oldest unacked msg */
        byte    m;              /* seq# last msg received  */
        byte    s;              /* seq# next msg to send   */
        byte    window;         /* nr of outstanding msgs  */
        byte    type;           /* msg type                */
        bit     received[M];    /* receiver housekeeping   */
        bit     x;              /* scratch variable        */
        byte    p;              /* seq# of last msg acked  */
        byte    I_buf[M], O_buf[M];     /* message buffers */
```

The `fc` model contains code for independent sender and receiver actions in full-duplex communications. Some of the housekeeping, then, has to do with keeping track of outgoing messages and some with incoming messages on the return channel.

Outgoing messages are stored in a message buffer `O_buf`, indexed by their sequence number. Buffering the outgoing messages allows the protocol to retransmit old messages when they are not acknowledged. A boolean array `busy` is used to remember which slots in the array of outgoing data are free and which are taken.

The main body of the flow control layer is a single `do` loop that checks for outgoing messages from the session layer, adds sequence numbers, forwards the messages to the error control layer, and keeps track of their acknowledgment. In separate clauses

it checks for incoming messages from the error control layer, strips sequence numbers, and forwards the remainders to the session layer. Sending of messages can be modeled as follows. A message is only sent if it is available, i.e., if the message channel `ses_to_flow` is non-empty, and if the lower protocol layer is free to accept it, i.e., if the message channel `flow_to_dll` is non-full.

```
do
:: (window < W  && len(ses_to_flow[n]) >  0
                && len(flow_to_dll[n]) < QSZ) ->
        ses_to_flow[n]?type;
        window = window+1;
        busy[s] = true;
        O_buf[s] = type;
        flow_to_dll[n]!type,s;
```

There is a little extra dance to be done if the message sent is a message of type `sync`, which is used by the session layer to reset the flow control layer protocol. In that case, all busy flags are cleared, and the sequence number returns to zero.

```
        if
        :: (type != sync) ->
                s = (s+1)%M
        :: (type == sync) ->
                window = 0;
                s = M;
                do
                :: (s > 0) ->
                        s = s-1;
                        busy[s] = false
                :: (s == 0) ->
                        break
                od
        fi
```

When an `ack` message arrives, its sequence number `m` points to the message that is being acknowledged. The status of that message, kept in array `busy[m]`, is reset to zero, meaning that the slot has become free. In the receiver code discussed below this is included as follows:

```
:: dll_to_flow[n]?type,m ->
        if
        :: (type == ack) ->
                busy[m] = false
        ...
```

If the message that was acknowledged is the oldest outstanding message, the window can slide up one or more notches and make room for more messages to be transmitted. This is modeled with two independent conditional clauses in the outer execution loop. The advancement of the window is guarded with a timeout clause to protect against lost messages.

```
:: (window > 0 && busy[q] == false) ->
        window = window - 1;
        q = (q+1)%M
:: (timeout && window > 0 && busy[q] == true) ->
        flow_to_dll[n]!O_buf[q],q
```

All that remains is the modeling of the clauses for the receiver part of the flow control layer. The messages are received in a generic clause

```
:: dll_to_flow[n]?type,m ->
```

followed by a switch on the message type. Incoming data are buffered to protect against messages that are received out of order, e.g., as a result of message loss and retransmission, and to allow the flow control layer to forward these messages to the session layer in the right order. The boolean array `received` is used to keep track of which messages have arrived and which are pending.

Let us first look at the processing of `sync` and `sync_ack` messages.

```
if
...
:: (type == sync) ->
        m = 0;
        do
        :: (m < M) ->
                received[m] = 0;
                m = m+1
        :: (m == M) ->
                break
        od;
        flow_to_dll[n]!sync_ack,m
:: (type == sync_ack) ->
        flow_to_ses[n]!sync_ack,m
```

The `sync` message is meant to initialize the flow control layer. In this case, the message comes from the remote peer and should be acknowledged with an `sync_ack` when the re-initialization is completed. If an `sync_ack` message arrives from the remote peer, it is passed on to the session layer protocol. The `sync_ack` message echos the session number of the `sync` message (cf. page 150, bottom).

All other messages are considered to be data messages:

```
:: (type != ack && type != sync && type != sync_ack)->
        if
        :: (received[m] == true) ->
                x = ((0<p-m   && p-m<=W)
                ||   (0<p-m+M && p-m+M<=W));
                if      /* ack was lost? */
                :: (x) -> flow_to_dll[n]!ack,m
                :: (!x) /* else skip */
                fi
```

```
                        :: (received[m] == false) ->
                                I_buf[m] = type;
                                received[m] = true;
                                received[(m-W+M)%M] = false
                        fi
```

When a data message arrives, we must first check whether or not it was received before (cf. page 79). If it was received before, we have `received[m]==true`, and we must check to see if it has been acknowledged yet. Messages are only acknowledged after they have been passed on to the session layer and have freed up the buffer space that they held. A message that has not been received before is stored, and the appropriate flag is set in array `received`.

The same clause is used to reset the received flag to false for the message that is one full window size away from the last received message: only at this point in the protocol can we be certain that this message cannot be transmitted again. The acknowledgment for that message must have been received or we could not have received the current message.

If the current message was received before, a check on the sequence number tells us if it was previously acknowledged or not. If it was, the fact that it was retransmitted indicates that the acknowledgment was lost, and needs to be repeated. One more clause remains: the one that encodes the accept process from Figure 4.12.

```
        :: (received[p] == true && len(flow_to_ses[n])<QSZ
                                && len(flow_to_dll[n])<QSZ) ->
                flow_to_ses[n]!I_buf[p];
                flow_to_dll[n]!ack,p;
                p = (p+1)%M
        od
}
```

The flow control layer is now complete. It takes some time to convince ourselves that it really works. Though the protocol is only about a hundred lines of code, the behavior it specifies can be complex, especially in the presence of transmission errors.

Correctness Requirements: The main correctness requirement for this protocol layer is that it transfers messages without deletions and reorderings, despite the behavior of the underlying transmission channel. To express, or check, this requirement we could label each message transferred by the sender, and check at the receiver that no labels are lost and that the relative ordering of the labels is undisturbed. In a way, such a label acts like just another sequence number.

Given a flow control protocol with a window size `W` and a range of sequence numbers `M`, how many distinct labels would minimally be needed to check the correctness requirement? The answer, due to Pierre Wolper (see Bibliographic Notes), is surprising. The number is independent of `W` and `M`: three different labels suffice for any protocol. Consider the following checking experiment. We transmit sequences of messages through the flow control layer, carrying just the label, and no other data. The three different types of labels are arbitrarily called `red`, `white`, and `blue`. For convenience we call the corresponding messages `red`, `white`, and `blue` messages as

well. To construct a range of test sequences, one `red` and one `blue` message are placed randomly in an infinite sequence of `white` messages. If the flow control protocol can ever lose a message, it will be able to lose the `red` or the `blue` message in at least one of the test sequences. Similarly, if the protocol can ever reorder two messages, it will be able to change the order of the `red` and the `blue` message in at least one sequence. The test sequences can be generated by a fake session layer protocol module.

```
proctype test_sender(bit n)
{       do
        :: ses_to_flow[n]!white
        :: ses_to_flow[n]!red -> break
        od;
        do
        :: ses_to_flow[n]!white
        :: ses_to_flow[n]!blue -> break
        od;
        do
        :: ses_to_flow[n]!white
        :: break
        od
}
```

The matching test model, which receives the test messages at the remote end of the protocol, can be defined as follows.

```
proctype test_receiver(bit n)
{       do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> break
        od;
        do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?blue -> break
        od;
        do
        :: flow_to_ses[n]?white
        od
}
```

The correctness requirement can now be formalized with a temporal claim or even more simply by adding some assertions to the receiver process.

```
proctype test_receiver(bit n)
{       do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> break
        :: flow_to_ses[n]?blue -> assert(0)     /* error */
        od;
```

```
        do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> assert(0)      /* error */
        :: flow_to_ses[n]?blue -> break
        od;
        do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> assert(0)      /* error */
        :: flow_to_ses[n]?blue -> assert(0)     /* error */
        od
}
```

This completes the design of the three main protocol layers. We have designed a protocol model that has sufficient detail to allow us to express and verify a set of formal correctness requirements about the protocol. It is not an implementation. The issue, however, need not come up until after the design itself has been validated (Chapter 14). Only then, the task of deriving an efficient implementation from the design will become relevant. We only take a quick peek at some of the issues here.

AN ASIDE ON IMPLEMENTATION

It is not relevant to the design of the validation model, but just out of curiosity we can look at how we could implement the parsing of messages at the lowest level in the protocol, just below the error control layer, or perhaps even combined with it. In C, for instance, the bytes obtained from the line can be stored, without processing, in a buffer `raw` that is large enough to hold one complete data message. We assume that the line drivers take care of character stuffing and de-stuffing, so we have a pure message delimited by `STX` and `ETX` control bytes.

If we use the protocol with the fields in the message header rounded to the nearest multiple of eight (Table 7.1), the maximum buffer size required is 428 bytes (422 data bytes plus 6 bytes overhead). The bytes are read into the buffer in a tight loop that can be coded as follows. We use a `union` of a raw buffer, a data message, and a non-data message. The code below assumes that the low-order byte of a 16-bit number, such as the checksum, is sent before the high-order byte.

```
typedef struct DATA_MSG {
        unsigned char type;
        unsigned char seqno;
        unsigned char tail[424];        /* includes checksum */
} DATA_MSG;

typedef struct NON_DATA_MSG {
        unsigned char type;
        unsigned char seqno;
        unsigned char checksum[2];
} NON_DATA_MSG;
```

```
union {
        unsigned char   raw[426];       /* data+header+trailer */
        DATA_MSG        data;
        NON_DATA_MSG    non_data;
} in;
```

Assuming that protection against the accidental occurrence of the STX and ETX message delimiters is provided by the sender by inserting (stuffing) a DLE (data link escape) character before all data that matches one of the three special characters STX, ETX, or DLE, we can write the byte scanning and de-stuffing routine as follows.

```
recv()
{       unsigned char c;
        unsigned char *start = in.raw;
        unsigned char *stop = start+428;
        unsigned char *ptr  = start;
scan:
        for (;;)                                /* forever   */
        {       if ((c = line_in()) == STX)     /* next byte */
                        ptr = start;            /* reset ptr */
                else if (c == ETX)
                        goto check;             /* have msg  */
                else if (ptr < stop)
                {       if (c == DLE)           /* escape    */
                                *ptr++ = line_in();
                        else
                                *ptr++ = c;     /* store     */
                }
        }
check:
        ...
}
```

When the ETX marker has been seen we must check for a valid message by calculating the checksum, for instance by calling the CRC routine discussed in Chapter 3. If indeed the low-order byte of the checksum is sent first, a recalculation of the checksum of all the raw data as received, *including* the checksum field, should come out zero in the absence of transmission errors:

```
check:
        if (cksum(in.raw, ptr-start) == 0)
                /* good message, accept */
        else
                /* distorted message, ignore */
        goto scan;      /* resume scanning for messages */
```

When the checksum is nonzero, the buffer contents must be discarded; otherwise it can be copied to a safe place and passed to the upper protocol layer.

7.7 SUMMARY

In this chapter we have discussed a protocol design, leading up to a hierarchy of validation models written in PROMELA. The function of each layer in this hierarchy is largely independent of the functions performed by the other layers. Each layer adds some functionality, and helps to transform the underlying physical channel into a virtual one with a progressively more idealized behavior.

With the derivation of the validation models the design process is not complete. The behavior of the model will have to be formally validated, perhaps revised, and finally implemented. Finding unspecified receptions, unexecutable code segments, or deadlocks is almost impossible to do by inspection alone. The normal, expected sequences of events can be checked easily. The errors, however, usually hide in the unexpected combinations of events: execution sequences that have a low probability of occurring. In Chapter 14, we consider the validation of the models we have derived.

If the implementation is not derived automatically from the validated model, we must add another validation step to guarantee that the implementation and the model are equivalent: conformance testing (see Chapter 8, Section 8.6, and Chapter 9).

EXERCISES

7-1. Compare the protocol hierarchy derived in this chapter to the OSI reference model discussed in Chapter 2, Section 2.6. Which layers are missing? □

7-2. Describe precisely in which order framing (using `STX` and `ETX` symbols), byte stuffing, and checksum calculations must be performed at the receiver and at the sender. □

7-3. Which layer would have to be modified to include a rate control method in this protocol? Which layer would have to be modified to include a dynamic flow control method? Describe a simple version of each method. □

7-4. Consider the possibility of recalculating the optimal data size `D` at run time and using the information gained to make the protocol adapt to dynamically changing channel characteristics. □

7-5. Complete the state transition diagram from Figure 7.8, adding transitions and states for all messages in the protocol vocabulary. □

7-6. Redesign the protocol for a transmission rate of 2400 bps. □

7-7. Redesign the protocol for a channel with zero error rate and zero signal propagation delay. □

7-8. (Doug McIlroy) Extend the protocol to allow for the file transfer to resume where it left off after carrier loss. Consider the option to use a status parameter in the accept and reject messages sent by the presentation layer to indicate how much of the file was successfully transferred. Hint: there can be no acknowledgment for the last data received and stored at the remote system at the moment of carrier loss. □

7-9. Is there a possibility that the two parties in the file transfer protocol can be caught in an execution cycle, or *livelock*, where both continue to send messages, but neither party succeeds in reaching the data transfer phase? If true, suggest a way to amend the protocol to avoid this problem. □

7-10. Redesign the file server process to allow for asynchronous communication with the session layer. □

7-11. Try to merge the two data link layers into one layer, performing both flow control and error control. □

7-12. Show that the one-bit `toggle` session number protects against the misinterpretation of `sync_ack` messages. Hint: compare with the alternating bit protocol. □

7-13. Suppose that the channel assumptions are changed to include the possibility of data reordering on the transmission channel. Does the one-bit connection set up method from Exercise 7-12 still work? Hint: a standard solution to this generalized version of the problem is known as the *three-way handshake*, see for instance, Stallings [1985, p. 490]. □

7-14. Consider what would need to be changed in the implementation (see ''An Aside on Implementation'') for transmissions in which the high-order byte of a 16-bit quantity is sent before the low-order byte. □

7-15. Express the correctness requirement for the flow control layer in a temporal claim. Hint: first express a requirement on the correct behavior; then reverse it to specify all invalid behaviors. Use a do-loop labeled with an acceptance-state label as the specification of the error state that is reached (and never exited) whenever the correct behavior is violated. □

7-16. Check the ten design rules from Chapter 2 and verify how they were applied to this design. Criticize the design where the rules were violated. □

BIBLIOGRAPHIC NOTES

Software layering is a concept that is primarily due to E.W. Dijkstra (see Bibliographic Notes, Chapter 2). More on the derivation of protocol parameters, such as the optimal data length, can be found in Field [1976], Tanenbaum [1981, 1988], or Arthurs, Chesson and Stuck [1983].

In spirit at least, the design borrows many ideas from the world of light-weight protocols (see Chapter 2). The importance of a judicious placement of control information in either the header or the trailer of a protocol, was first emphasized by Greg Chesson, one of the initiators of light-weight protocol design.

In many standards all control information is by default placed in one single control block that is placed at the header of each message. The term ''trailer protocol'' was coined by Chesson in an effort to show that the opposite discipline, of placing all control information in the trailer and none, except perhaps a routing address, in the header, could lead to more efficient encodings of both sender and receiver. For instance, both the byte count and the checksum of a message can then be computed on the fly by the sender. The receiver can similarly compute checksums on the fly (starting at an `STX` control flag), and after spotting the matching `ETX` flag the receiver can find the byte count close by. The checksum can be verified immediately.

The correctness criterion that we specified for the flow control layer was first discussed in Wolper [1986]. It is also discussed in Aggarwal, Courcoubetis, and Wolper [1990].