FLOW CONTROL 4

4.1 INTRODUCTION

The simplest form of a flow control scheme merely adjusts the rate at which a sender produces data to the rate at which the receiver can absorb it.[1] More elaborate schemes can protect against the deletion, insertion, duplication, and reordering of data as well. But let us first look at the simpler version of the problem. It is used

- To make sure that data are not sent faster than they can be processed.
- To optimize channel utilization.
- To avoid data clogging transmission links.

The second and the third goals are complementary: sending the data too slowly is wasteful, but sending data too fast can cause congestion. The data path between sender and receiver may contain transfer points with a limited capacity for storing messages shared between several sender-receiver pairs. A prudent flow control scheme prevents one such pair from hogging all the available storage space.

In this chapter we build up a full flow control discipline in a sequence of modifications of a simple, basic model. The procedure rules of these protocols are specified with the flow charting language introduced in Chapter 2 and summarized in Appendix B. The notation *mesg:o* in an input or output statement, for instance, indicates that a message of type *mesg* with data field *o* is received or sent, respectively. The statement *next:o* indicates the internal retrieval of data item *o* to be transmitted in the next output message. Similarly, *accept:i* indicates the acceptance (storage) of *i* as correctly received data.

Figure 4.1 illustrates a protocol without any form of flow control. Note that it is a *simplex* protocol: it can be used for transfer of data in only one direction (see Figure 2.1 and Appendix A).

---

1. At the lowest level such synchronization must already take place to drive a physical line. See *Synchronous and Asynchronous Transmission* in Appendix A.

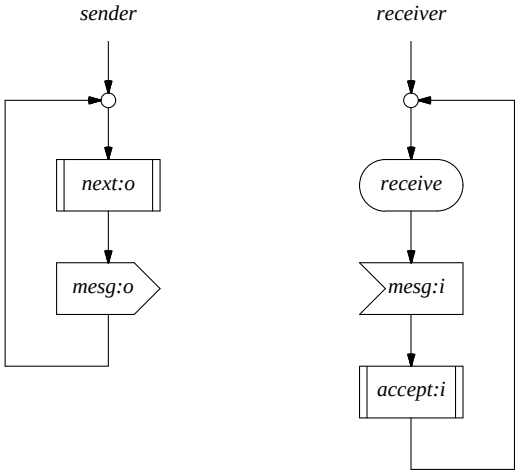

Figure 4.1 — No Flow Control

The protocol in Figure 4.1 only works reliably if the receiver process is guaranteed to be faster than the sender. If this assumption is false, the sender can overflow the input queue of the receiver. The protocol violates a basic law of program design for concurrent systems:

Never make assumptions about the relative speeds of concurrent processes.

The relative speed of concurrent processes depends on too many factors to base any design decisions on it. Apart from that, the assumption about the relative speed of sender and receiver is often not just dangerous but also invalid. Receiving data is generally a more time-consuming process than sending data. The receiver must interpret the data, decide what to do with it, allocate memory for it, and perhaps forward it to the appropriate recipient. The sender need not find a provider for the data it is transmitting: it does not run unless there are data to transfer. And, instead of allocating memory, the sender may have to free memory after the data are transmitted, usually a less time-consuming task. Therefore, the bottleneck in the protocol is likely to be the receiver process. It is bad planning to assume that it can always keep up with the sender.

The oldest and least reliable flow control technique that can be used to address this synchronization problem requires no prior negotiation between sender and receiver about the pace at which messages can be transmitted. The method uses two control messages: one to *suspend* and one to *resume* traffic. The messages are sometimes called *x-off* and *x-on*.[2] Assume, then, that we have an error-free channel and a protocol vocabulary of the following three message types:

V = { mesg, suspend, resume }

2. The *control-s* and *control-q* codes on many data terminals provide the same two functions.

where the control messages *suspend* and *resume* are used to implement the flow control discipline. The procedure rules of the protocol can now be added. We implement them here with two additional processes, one in the sender and one in the receiver, as shown in Figure 4.2.

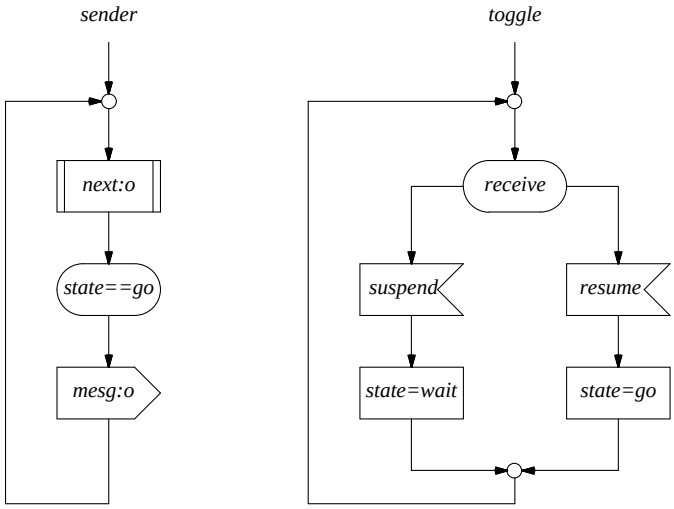

Figure 4.2 — X-on/X-off Protocol: Sender Processes

After receiving a *suspend* message, the *toggle* process in the sender sets the value of a variable *state* to *wait*. It resets the variable to its initial value *go* after the arrival of a *resume* message. The sender process simply waits (at the oval box) until *state* has the proper value before transmitting the next message.[3] 3. Recall that the oval box indicates a potential delay. The process waits for a message to arrive when the box is labeled *receive*, or else it waits until the condition specified becomes true. Cf. Figure 2.1.

The receiver is also split into two parts. After the arrival of a data message a counter

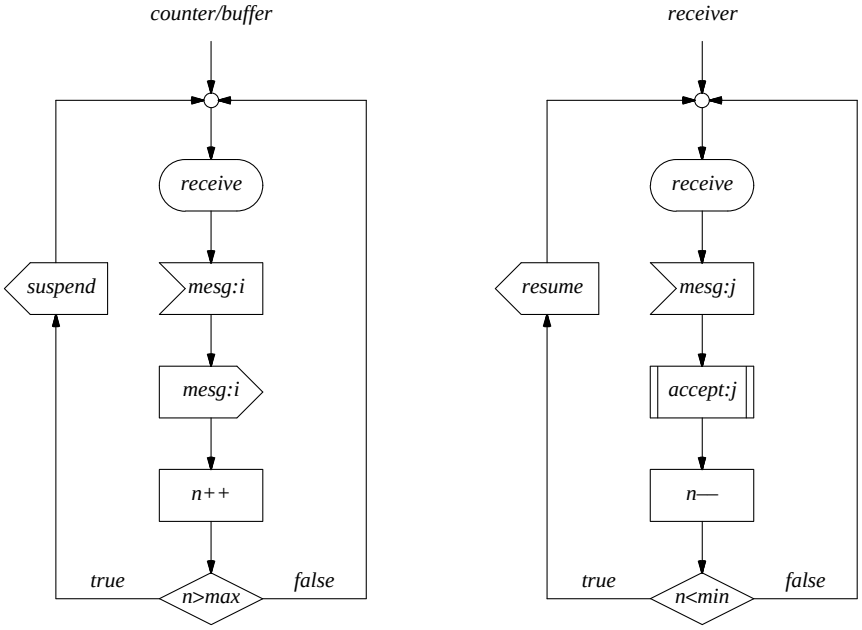

Figure 4.3 — X-on/X-off Protocol: Receiver Processes

process increments a variable *n*, and upon the acceptance of the message, an acceptor process decrements it. The data messages are passed from the counter process to the acceptor process via an internal queue. The count remembers the number of messages that have been received from the sender and the number that are waiting to be accepted by the receiver. If its value increases beyond some predefined limit, a *suspend* message is sent to the sender. If it drops below a lower bound, the *resume* message is sent, as shown in Figure 4.3. To split the receiver into two processes, of course, only makes sense if *accept* is a relatively time-consuming operation.

There are some problems to be resolved. The correct working of the protocol depends on the properties of the transmission channel. If a *suspend* message is lost or even delayed, the overflow problem recurs. The working of a protocol should not depend on the time it takes a control message to reach the receiver. Worse still, if a *resume* message is lost, the four-process system comes to a complete halt.

We have these two problems to solve:

- Protect against overrun errors in a more reliable way.
- Protect against message loss.

A standard method of solving the first problem is to let the sender explicitly wait for the acknowledgment of transferred messages. An example is the *Ping-Pong* protocol of Figure 4.4. This method is often called a *stop and wait* protocol. The overflow problem has disappeared, but the system still deadlocks if either a control or a data message is lost. The sender and receiver are too tightly coupled. Let *t* be the message

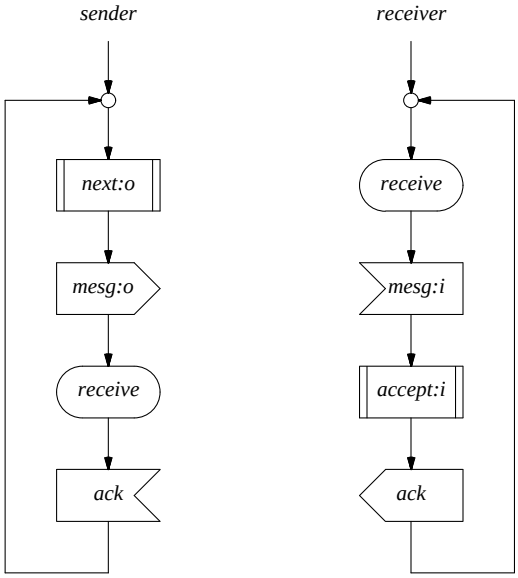

Figure 4.4 — Ping-Pong Protocol

propagation time on the channel, a the time it takes the receiver to process and accept an incoming message, and p the time it takes the sender to prepare a message for transmission. With the above scheme the sender incurs a delay of $2t + a - p$ units of time for every message transmitted.

Typically $p < a$ and, obviously, t increases at least linearly with the distance between sender and receiver. Note, however, that the acknowledgment message does not just signify the arrival of the last message, it is also used as a *credit* that the receiver extends to the sender to transmit the next message. This idea directly leads to a solution that can alleviate the delay problem: the window protocol.

4.2 WINDOW PROTOCOLS

In a call-setup phase, the receiver can tell the sender exactly how much buffer space it is prepared to reserve for incoming messages. The sender is then given *credit* for a fixed number of outstanding messages. The credit can be updated dynamically when the amount of available buffer space changes.

Let us not worry about message loss just yet and first look at the basic working of a window protocol. Each message received is acknowledged with a single *ack* control message on a return channel. All we have to do is to keep count of the number of messages in transit.

The initial credit can either be negotiated, or it can be set to a fixed number of messages W. For each message sent the sender decrements its credit, and for each message received the receiver extends a new credit to the sender via the return channel.

The example protocol shown in Figure 4.5 illustrates this. The quantity $W-n$ gives the number of unused credits.

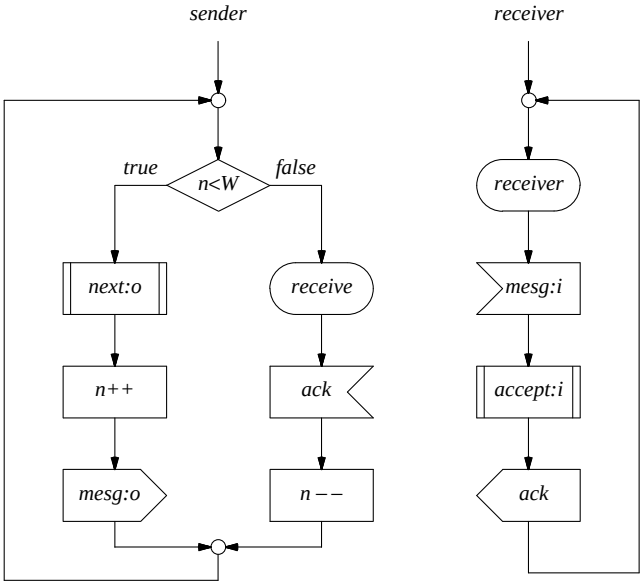

Figure 4.5 — Window Protocol for an Ideal Channel

Let $a(t)$ be the number of credit messages received by the sender at time t after initialization, let $m(t)$ be the number of messages sent to the receiver, and let $n(t)$ be the value of n at time t. The maximum number of messages that the sender can have outstanding, waiting acknowledgment, is

$$W-n(t)+m(t)-a(t)$$

where $W-n(t)$ is the number of unused credits and $m(t)-a(t)$ the number of used credits. We would like to convince ourselves that

$$W-n(t)+m(t)-a(t)\leq W$$

or

$$m(t)-a(t)\leq n(t)$$

Initially all variables in this inequality are zero and the condition is trivially true. Every send action in the sender increments both sides of the inequality, right side first, and preserves its validity. Similarly, with every receive action the receiver process decrements both sides by one, the left side first, again preserving the correctness.

MESSAGE LOSS

The maximum credit W is called the *window size* of the protocol. During a transfer, the current credit varies between zero and W, depending on the relative speeds of

sender and receiver. The sender is only delayed when the credit is reduced to zero. This flow control discipline can optimize communication on channels with long transit delays by enabling the sender to transmit new messages while waiting for the acknowledgment of old ones.

The problems of lost, inserted, duplicated, or reordered messages do, of course, still exist. If, for instance, a set of acknowledgment messages is lost, both parties may hang: the sender waiting for the acknowledgments that were lost, the receiver waiting for the messages it credited.

TIMEOUTS

To protect against the loss of essential messages the sender has to keep track of elapsed time. In the Ping-Pong protocol of Figure 4.4, for instance, the sender can try to predict the worst turn-around time for each acknowledgment. If the response has not arrived within that period, the sender can *time out* and assume that it was lost.

In practice, the ''worst'' turn-around time is often calculated with a heuristic:

$$T_{worst} = \overline{T} + N \cdot \sqrt{var(T)}$$

where T is the round-trip delay N is usually one, and rarely larger than two. The round-trip delay is simply the time it takes a message to go from sender to receiver plus the time it takes a response to return to the sender (see Exercise 4-12). $\overline{T}$ and $var(T)$ are, respectively, the *average* and the *variance* of T. The factor N is thus a multiplication factor for standard deviation of the turn-around time (the square root of the variance).

In many cases, the behavior of the receiver process at the far end of a transmission channel can be modeled by an M/M/1 queueing system.[4] We then assume that, from the receiver's point of view, the distribution function of the interarrival times of messages is a Poisson process and the distribution time for the processing of these messages is a simple exponential function. For an M/M/1 queueing system, it can be shown that the variance of the time spent in the system is the square of the mean. This means that for our transmission channel the variance of both the one-way and the round-trip delay is also the square of the mean, $var(T) = \overline{T}^2$. This leads to the simple rule of thumb that an approximation for the retransmission time can be obtained by doubling the average round-trip delay T (assuming a factor $N = 1$ in the above estimate:

$$T_{worst} \approx 2 \cdot \overline{T}$$

A timeout after a deletion error certainly looks straightforward. A common mistake, however, is to let both the sender *and* the receiver use timeouts. Consider the extension of the Ping-Pong protocol shown in Figure 4.6.

---

4. The notation is due to D.G. Kendall [1951].

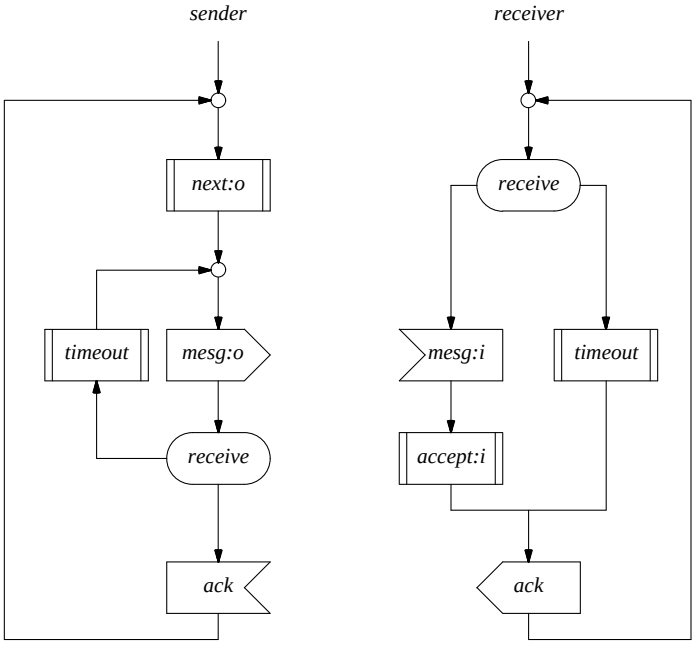

Figure 4.6 — Ping-Pong Protocol with Timeouts

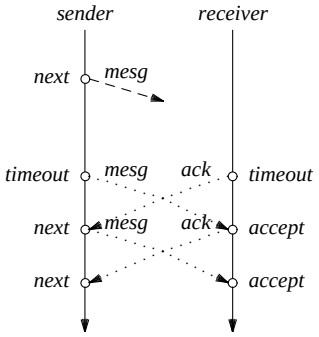

Figure 4.7 — Time Sequence Diagram of An Error

Figure 4.7 shows what can happen with this protocol if a deletion error occurs. Both sender and receiver decide to retransmit the last sent message when a deletion error is assumed. When the first *ack* message reaches the sender, it cannot possibly tell whether it acknowledges the lost or the retransmitted message. The sender ends up matching the wrong *ack* and *mesg* messages indefinitely.

One lesson to be learned from this is that sender and receiver should not both be able to initiate retransmissions. It is sufficient to place this responsibility with one of the two processes. Traditionally, this is the sender process, since only the sender can

know for certain when new data has been sent. Another lesson is that we must be able to tell from an acknowledgment exactly which message is being acknowledged, even if we only intend to send one message at a time, as in the Ping-Pong protocol. We can do this by adding *sequence numbers* to each data and control message. By doing so, we also obtain a mechanism for solving other classes of transmission problems in a fairly straightforward way: duplication errors and out-of-sequence messages.

Since sequence numbers necessarily have a restricted range,[5] we must have a way to verify that recycled numbers cannot disturb the correct working of the protocol. We will see below that if sequence numbers are used in combination with a window protocol this requirement can be fulfilled relatively easily. Before we make that combination, the *sliding* window protocol, let us take a closer look at the use of timeouts and sequence numbers.

4.3 SEQUENCE NUMBERS

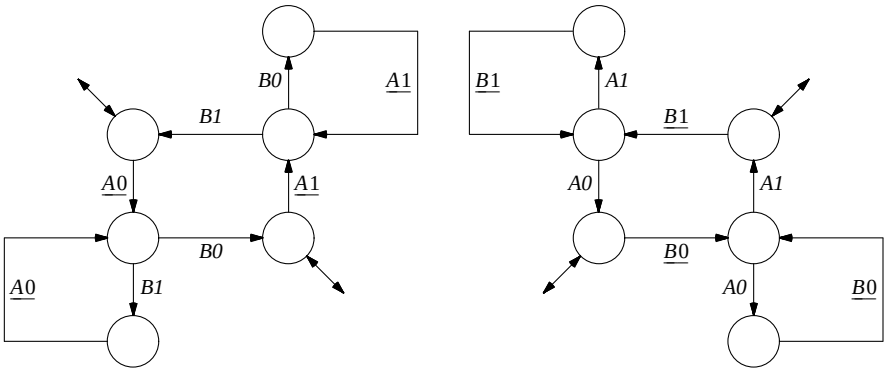

Figure 4.8 — Original Alternating Bit Protocol

As an example of a better use of a timeout, and a one-bit sequence number, we can consider an extended version of the alternating bit protocol (a famous protocol, see the Bibliographic Notes). The protocol continues to surface in so many different disguises in the protocol literature that it is worthwhile to first look at the original specification from Bartlett, Scantlebury and Wilkinson [1969]. In their paper, the protocol is defined with two finite state machines of six states each, as shown in Figure 4.8. The original protocol, therefore, can be in no more than 36 different states, substantially fewer than all other variations that have been studied.

Figure 4.8 specifies the behavior of two processes, *A* and *B*. The notation is from Bartlett, Scantlebury and Wilkinson [1969]. The edge labels specify the message exchanges. Each label consists of two characters. The first specifies the origin of the message being received or transmitted, and the second specifies the sequence number,

5. There is only a finite number of bits to store them in the message headers.

called the *alternation bit* in the original paper. Send actions are underlined.

The double headed arrows indicate states where input is to be accepted in the receiver or where a new message is fetched for output in the sender. Erroneous inputs, i.e., messages that carry the wrong sequence number, prompt a retransmission of the last message sent. It is relatively easy to extend the protocol with timeouts to allow for recovery from message loss. A flow chart version of this extension is shown in Figure 4.9.

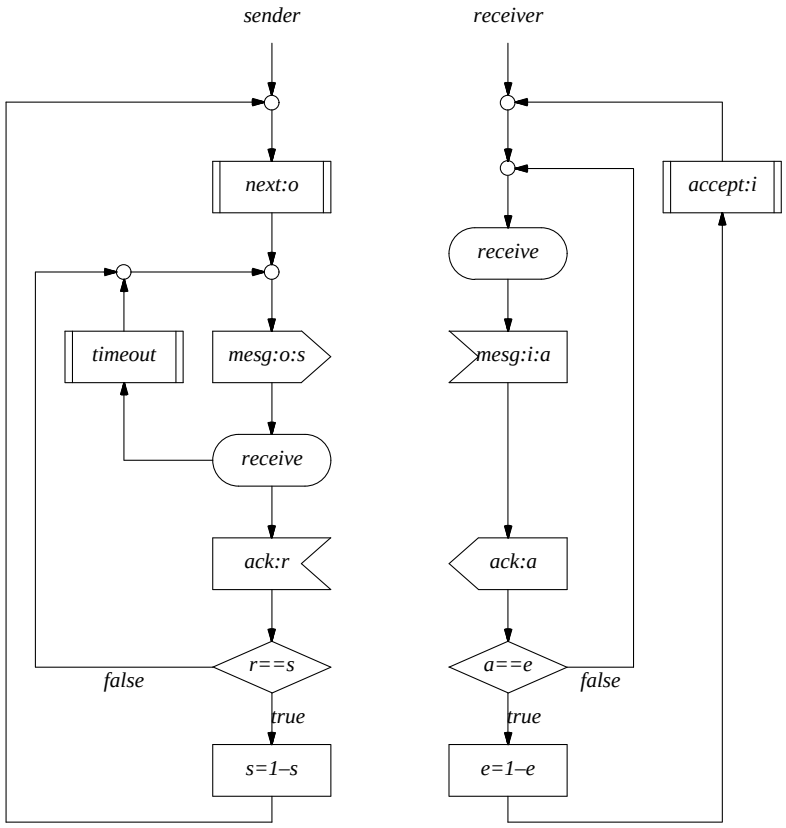

Figure 4.9 — Alternating Bit Protocol with Timeouts

We have used two types of messages, *mesg* and *ack*, with, for instance, the format

{ mesg, data, sequence number }

and

{ ack, sequence number }

respectively. In the flow chart, *mesg:o:s* indicates a message *mesg* with data field *o* and sequence number field *s*.

We have also used four single-bit variables: *a*, *e*, *r*, and *s*. Variable *s* is used by the

sender to store the last sequence number sent, and *r* holds the last sequence number received. The receiver uses *e* to hold the next number expected to arrive and variable *a* to store the last actual sequence number received. All variables have an initial value zero.

Figure 4.10 illustrates what happens if the deletion error from Figure 4.7 occurs in the alternating bit protocol. The protocol recovers from the error when the sender process times out and retransmits the lost message.

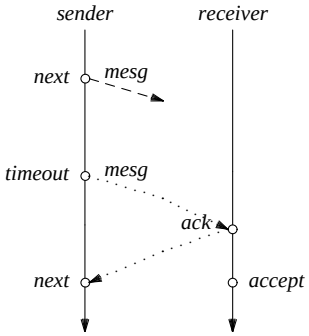

Figure 4.10 — Time Sequence Diagram of Error

Consider also what happens if an acknowledgment is delayed long enough for the sender to time out and retransmit the last message (see Exercise 4-6).

MESSAGE REORDERING

Now let us consider the duplication and reordering of messages, as may happen in, for instance, datagram networks where messages can travel along different routes to their destination. The obvious solution is to encode the original order of the messages in a larger sequence number that is attached to each message. With a 16-bit field for the sequence numbers we can number 65,536 subsequent messages. Assuming a message length of 128 bits and an effective line speed of 9600 bps (bits per second), we could run out of numbers within 15 minutes. Fortunately, this range problem readily disappears if we limit the maximum number of messages that can be in transit at any one time: the sender's credit. Clearly, the range of the sequence numbers has to be larger than the maximum credit used so that a receiver can always distinguish duplicate messages from originals.

Assume a range *M* of available sequence numbers and an initial credit of *W* messages. We assume for the time being that *M* is sufficiently larger than *W* to avoid confusion of recycled sequence numbers. The sender must do some bookkeeping for every outstanding message within the current window. We use two arrays for this purpose. Boolean array element *busy[s]* is set to *true* if a message with sequence number *s* was sent and has not yet been acknowledged. The second array *store[s]* remembers the last message with sequence number *s* that was transmitted. Initially, all elements of array *busy* are set to *true*.

There are many problems to solve to get this version of the window protocol to work. The task can be split into three subtasks: transmitting messages, processing acknowledgments, and retransmitting messages that remain unacknowledged for too long. In addition to the constants *W* and *M*, the following four variables are used, all with an initial value of zero:

- *s*, the sequence number of the next message to send
- *window*, the number of outstanding unacknowledged messages
- *n*, the sequence number of the oldest unacknowledged message
- *m*, the sequence number of the last acknowledged message

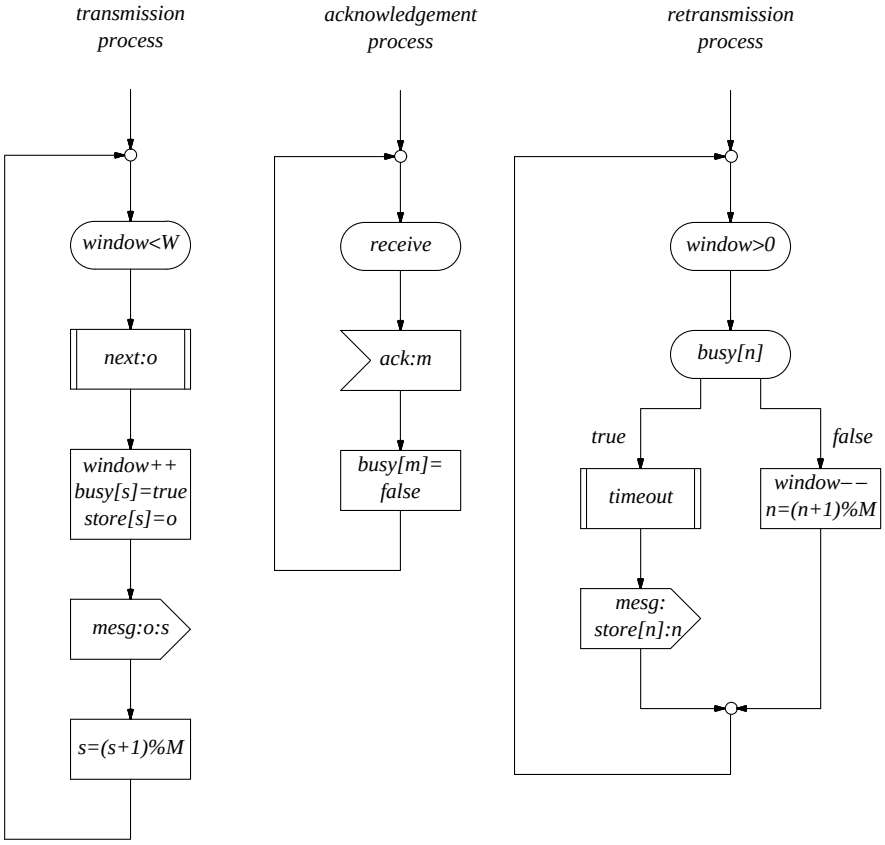

Figure 4.11 — Sender Processes, Sliding Window Protocol

First consider the transmission process in Figure 4.11. As long as all credits have not been used up, messages can be transmitted. Each message transmitted increments the number of outstanding messages, and by doing so, it implicitly decrements the credit for the transmission of new messages. A sequence number *s* is assigned, the message contents are stored in *store[s]* for possible retransmission later, the flag is set in

busy[s], and *s* is incremented modulo the range of the sequence numbers *M* (using the '%' operator).

The acknowledgment process is even simpler. It receives the incoming acknowledgments and sets the *busy[m]* flag to *false*. The order in which these acknowledgments are received is irrelevant.

The retransmission process waits until there are messages in transit by checking that *window* is non-zero. Each message that is sent must ultimately be acknowledged and have its *busy[n]* flag reset to *false*. The retransmission process waits for this to happen at the second wait clause. If it does, the window size is decremented, and *n* is incremented to point to the next oldest unacknowledged message. If the *busy* flag is not reset to *false* within a finite amount of time, the retransmission process times out and retransmits the message. The oval box delays the process until the condition specified becomes *true* or, as in the current case, until a timeout occurs (cf. Appendix B). The way we have specified it here, the retransmission timer repeats just one message, the oldest unacknowledged message.

The receiver for the sliding window protocol is given in Figure 4.12. It is split into two processes. One process receives and stores the incoming messages in whatever order they may happen to arrive. A second process accepts and acknowledges the messages, using the sequence numbers to restore their proper order. Messages cannot be acknowledged until they are accepted, to avoid the risk of running out of buffers to store messages if the accepting process turns out to be slower than the sender. We use a boolean array *recvd[M]* to remember the sequence numbers of messages that have been received, but not yet accepted, and an array *buffer[M]* to remember the contents of those messages. There is one extra variable to keep track of the protocol's progress: *p*, the sequence number of the next message to be accepted. It has an initial value of zero.

The accept process is straightforward. It waits for the *received* flag of the next message to be accepted to become *true*, accepts and acknowledges the message, and increments *p*. The receiver checks whether a newly arrived message is an original or a duplicate. For a new message, the *received* flags are set, and the message is stored in array *buffer*. Two flags must be updated, one for the message that was just received and one for a message that we now know can no longer be received because it is outside the current window (see Exercise 4-14.)

$$recvd[m] = true$$

and

$$recvd[(m - W + M)\,\%\,M] = false$$

or equivalently

$$recvd[(m - W)\,\%\,M] = false$$

A duplicate message is recognized by the fact that the *received* flag was set to *true*

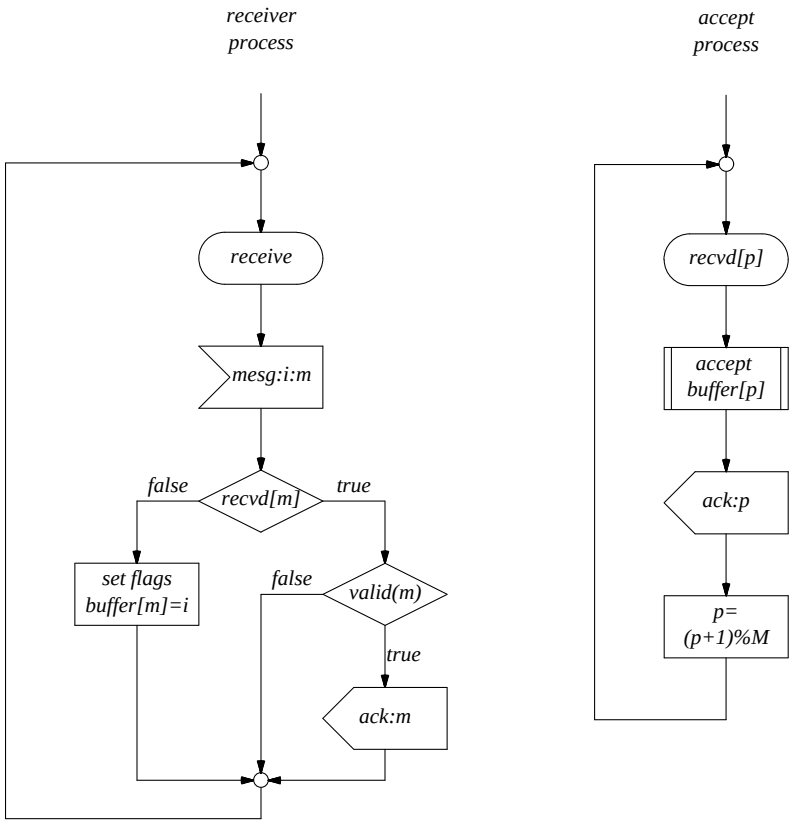

Figure 4.12 — Receiver Processes, Sliding Window Protocol

before. There are two possible reasons for the arrival of a duplicate message:

- The message was received, but not yet acknowledged.
- The message was received and acknowledged, but the acknowledgment somehow did not reach the sender.

Only in the second case should the acknowledgment be repeated. The current value of variable p should be sufficient to figure out which of the two cases applies. If the sequence number count was not modulo M, the test would simply be:

$$valid(m) = m < p$$

since only values smaller than p were acknowledged before. Taking the modulo M effect into account (p is always smaller than M), this becomes:

$$valid(m) = (0 < p - m \leq W) \;||\; (0 < p + M - m \leq W)$$

The window protocol guarantees that a retransmitted message cannot have a sequence number that is more than W smaller than the last message that was acknowledged.

The only case, then, where we can have $m > p$ or $p - m > W$ is when p has wrapped around the maximum M, and m has not.

MAXIMUM WINDOW SIZE

If M is the range of the sequence numbers, what is the maximum number of outstanding messages W that we can use and still guarantee that the window protocol works properly? If all messages that arrive out of order were simply rejected by the receiver, the answer would be $M - 1$. As long as a sequence number is not recycled before the last message using it is acknowledged, all is well. This means that if messages may be received out of order, as in Figure 4.12, the window size cannot exceed $M/2$ (cf. Exercise 4-9).

As an example, consider the following case. Let H be the highest sequence number (modulo M) that the receiver has read and acknowledged. It signifies to the receiver that the sender has at least processed an acknowledgment for the W-th message preceding the one numbered H (observation 1). The receiver also knows that at best the sender has processed all acknowledgments up to and including the one for the message numbered H (observation 2).

- Observation 1 means that the sender may decide to retransmit any one of the $W - 1$ messages preceding H, and H itself. The oldest message that could be retransmitted would carry sequence number $(H - W + 1)\,\%M$.
- Observation 2 means that the sender may also transmit up to W of the messages that succeed the message numbered H. The first $W - 1$ of these messages may even be lost on the transmission channel so that the message with number $(H + W)\,\%M$ is the first new message to arrive.

The highest-numbered message that may succeed H must be distinguishable from the lowest-numbered message that may be retransmitted preceding sequence number H. This means $M > 2W - 1$, or a maximum window size of $W = M/2$.

4.4 NEGATIVE ACKNOWLEDGMENTS

So far, we have used acknowledgments as a method of flow control, not of error control. If a message is lost or damaged beyond recognition, the absence of a positive acknowledgment would cause the sender eventually to time out and retransmit the message. If the probability of error is high enough, this can degrade the efficiency of the protocol, forcing the sender to be idle until it can be certain that an acknowledgment is not merely delayed, but is positively lost. The problem can be alleviated, though not avoided completely, with the introduction of *negative* acknowledgments.

The negative acknowledgment is used by the receiver whenever it receives a message that is damaged on the transmission channel. How the receiver may be able to establish that is discussed in Chapter 3. When the sender receives a negative acknowledgment, it knows immediately that it must retransmit the corresponding message, without having to wait for a timeout. The timeout itself is still needed, of course, to allow for a recovery from messages that disappear on the channel.

Figures 4.13 and 4.14 show an extension of the alternating bit protocol from Figure

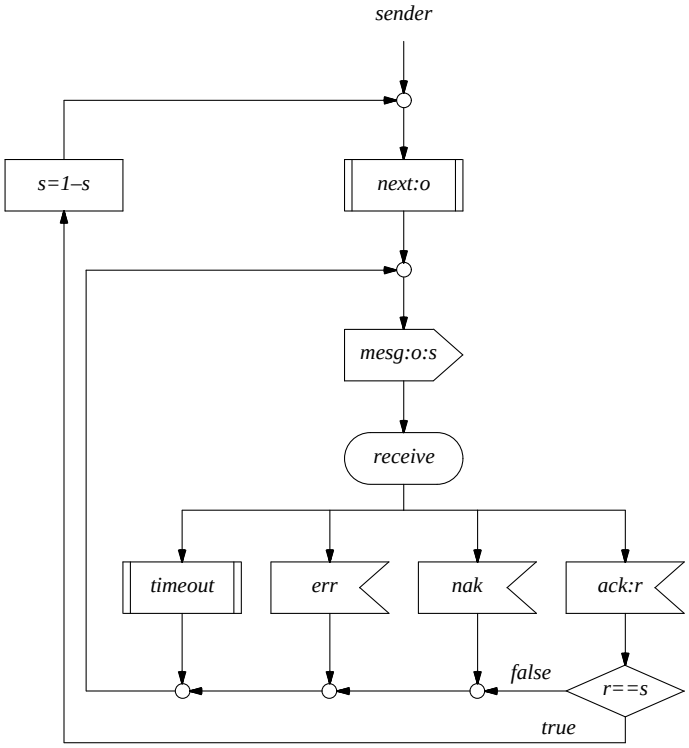

Figure 4.13 — Sender, Extended Alternating Bit Protocol

4.9 with negative acknowledgments. In this simple case, the *nak* needs no sequence number. (See also Exercise 4-3.)

TERMINOLOGY

The method of using acknowledgments to control the retransmission of messages is usually referred to as an *ARQ* method, where ARQ stands for Automatic Repeat Request. There are three main variants:

- Stop-and-wait ARQ
- Selective repeat ARQ
- Go-back-N continuous ARQ

The Ping-Pong protocol of Figure 4.4, possibly extended with negative acknowledgments, classifies as a *stop-and-wait ARQ*. After each message is sent, the sender must wait for a positive or a negative acknowledgment, or perhaps a timeout.

The use of acknowledgments in the sliding window protocol of Figures 4.11 and 4.12 is a *selective repeat ARQ* method. In Figure 4.11 implemented a ‘‘one-at-a-time’’ selective repeat method where only the oldest unacknowledged message is retransmitted. In general, however, any message that triggers either a negative acknowledgment or a timeout may be retransmitted, independently of any other outstanding message.

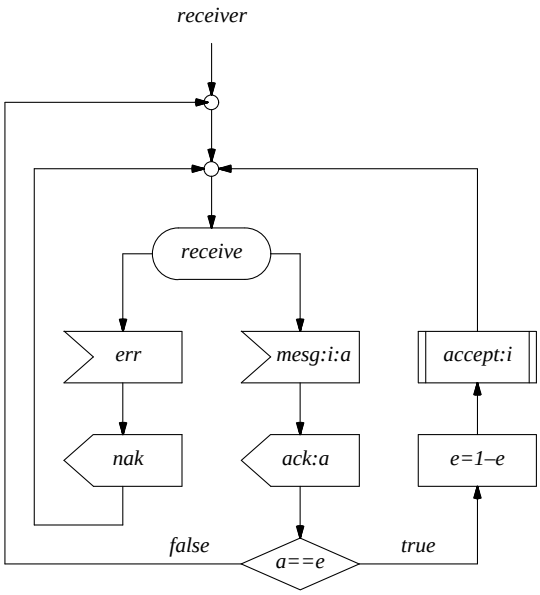

Figure 4.14 — Receiver, Extended Alternating Bit Protocol

The generalized method is called ‘‘continuous’’ selective repeat.

The last strategy, *go-back-N continuous ARQ*, could be implemented in the above protocol by having the sender retransmit the corrupted message and all subsequently sent messages. In that case the design of the receiver can be simplified. The accept processor from Figure 4.12, for instance, can now be deleted and the buffer becomes superfluous. In a go-back-N discipline the receiver refuses to accept all messages that arrive out of order, and waits for them to arrive in the proper sequence. It will not acknowledge any out-of-order messages. An acknowledgment with sequence number *s* can now be understood to acknowledge all messages *up to* and including *s*. Such an acknowledgment is therefore sometimes called a *cumulative acknowledgment*.

BLOCK ACKNOWLEDGMENT

A variation that can be used with the selective repeat and the go-back-N strategy to reduce the number of individual acknowledgment messages that must be sent from receiver to sender is known as *block acknowledgment*. In this case each positive acknowledgment can specify a range of sequence numbers of messages that have been received correctly. The block acknowledgment can be sent periodically or at the sender’s request. Block acknowledgment can be seen as an extended form of cumulative acknowledgment.

4.5 CONGESTION AVOIDANCE

At the start of this chapter we gave two main reasons for the inclusion of flow control schemes in protocols: synchronization and congestion avoidance.

Up to this point we have mostly ignored congestion avoidance and focused on end-to-end synchronization. One important issue in particular has not been discussed yet: For a given data link, how is the actual window size and the corresponding range of sequence numbers chosen? It is relatively easy to set an upper limit on the window size: at some point increasing it can no longer improve the throughput if the channel is already fully saturated.

Assume it takes 0.5 seconds for a message to travel from sender to receiver, and another 0.5 seconds for the acknowledgment to come back to the sender. The sender can then fully saturate the channel if it can keep sending data for 1 second. If the data rate of the channel is *S* bps the sender should be able to transmit *S* bits before it needs to check for acknowledgments. If there are *M* bits in each message that is transmitted, the best window size is trivially *S/M*. And, of course, we had better make certain that *M*<*S*. A larger window size than *S/M* is wasteful: by the time the last message in the current window is transmitted, the acknowledgment for the oldest outstanding message should have arrived, and if it has not, it may be time to start considering the retransmission of that message.

There is a danger in the type of calculation we have performed here. It reduces the flow control problem to a link-level issue, while ignoring the network that contains the data link. Consider, for example, the two-hop data link shown in Figure 4.15.

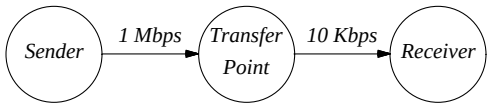

Figure 4.15 — Two-Hop Link

There are two ways of defining a flow control protocol for transfers from the sender to the receiver in this two-link network:

- Hop-by-hop (also called node-to-node)
- End-to-end

In a hop-by-hop protocol, the window size is calculated separately for each link to try to saturate each one. The first link is 100 times faster than the second. But if we succeed in saturating both channels we have only succeeded in creating a bigger problem. Data arrive at the transfer point about 100 times faster than they can be passed on to the receiver. No matter how much buffer space the transfer point initially has, it eventually runs out of space, and unless it can throttle down the sender, it will start losing messages.

The only way the transfer point can control the sender is to refuse to acknowledge messages. The sender, however, tries to saturate the channel and will do so, either

with retransmissions or with new data. If the number of acknowledgments drops, the sender will continue to saturate the channel by retransmitting data.

A flow control scheme, then, must be designed to optimize the utilization of two separate resources:

- The buffer space in the network nodes
- The bandwidth of the links connecting the nodes

The simple scheme above fails on both counts: it wastes buffer space in the transfer point, thereby potentially blocking other traffic that may be routed through that node, and it wastes bandwidth by triggering a deluge of retransmissions on the link from the sender to the transfer point. Optimal use of the two-link data path can only be achieved if the sender offers data at the data rate of the slowest link in the path: just 1% of the saturation point of the first link, which implies some type of feedback scheme from the second link back to the first.

In an end-to-end protocol this problem does not exist. The end-to-end capacity of the network path equals the capacity of the slowest link, and the window size can be set accordingly. The problem is that in a complicated network there is no hope that a sender can easily predict where the slowest link in its path to the receiver will be. The safest thing to do would be to derive a maximum window size for the whole network that is based on its slowest link. But that is hardly an inspiring solution, not to mention a wasteful one. Furthermore, in a larger network the capacity of a data link depends not just on the hardware but also on the number of competing users. If ten users start transferring large files over the fastest link in the network, that link can suddenly become the slowest one for all other users.

Going back to the original problem, even though we have pretended otherwise up to this point, flow control is not a static problem, but a dynamic one. In a static flow control protocol a sender always assumes that a message was either lost or distorted if its acknowledgment does not arrive with the round-trip message delay time. The appropriate response of the sender, in that case, is to retransmit the message. It can, however, also mean that the network is overloaded. The appropriate response of the sender is then to *reduce* the amount of traffic it offers to the network. The simplest method the sender has for doing this is to decrease its window size.

DYNAMIC FLOW CONTROL

A *dynamic window flow control* method makes the protocol self-adapting, one of the principles of sound design we listed in Chapter 2. A simple and commonly used method is to force a sender to decrease its window size whenever a retransmission timeout occurs. Once the timeouts disappear, the sender can be allowed to gradually increase the window size back to its maximum value. There are different philosophies about the precise parameters to be used in such a technique. Three popular variations are listed below.

- Decrease the window by *one* for every timeout that occurs, and increase it by one for *every* positive acknowledgment.
- Decrease the window to *half* its current size upon every timeout, and increase it by

one message for every *N* positive acknowledgments received.

- Decrease to its *minimum* value of one, immediately when a timeout occurs, and increase the window by one for every *N* positive acknowledgments received.

All methods assume a minimum window size of one. The maximum size can be calculated as before, or it can be set to a heuristic value, such as the number of hops on the link through the network between sender and receiver. The heuristic guarantees that in normal operation every intermediate node stores just one message per connection.

With all three techniques it is assumed that the protocol by default uses its precalculated maximum window size. The *slow start* protocol developed by Van Jacobson also removes that assumption: the protocol starts with the minimum window size of one, and only starts increasing the effective window size once the first acknowledgment has been received. In the slow start protocol the round-trip delay is continuously measured, and it, rather than the retransmission timeout, is used as a measure for increasing or decreasing the window size.

RATE CONTROL

With the dynamic window flow control schemes above, we have touched upon more specific network design issues, which are outside the range of this book. From a network operator's point of view, the best congestion avoidance technique is to control the amount of traffic that *enters* the network under overload conditions, rather than attempting to minimize the damage for the traffic that has already been accepted, for instance, with timeouts and retransmissions. These methods are collectively called *rate control methods*. Figure 4.16 shows a well-known throughput versus traffic load chart that illustrates the need for rate control.

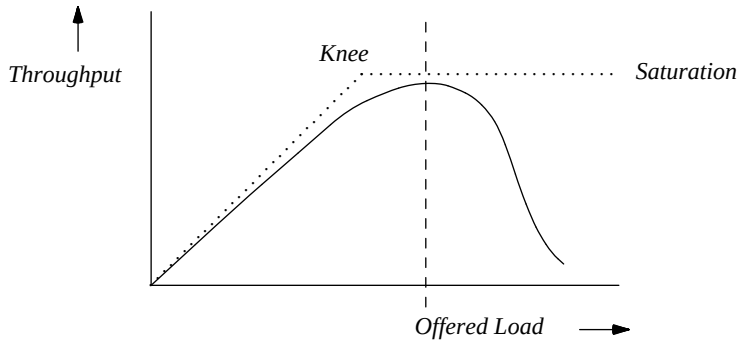

Figure 4.16 — Network Congestion

Ideally, the throughput of the network increases linearly with the offered load until it is fully saturated. In practice, network control algorithms eat away a little from the network capacity and a somewhat lower throughput is realized. Close to the saturation point, a growing offered load leads to an increasing degradation of service caused by the network congestion. The effect is comparable to a busy high-way where traffic

slowly comes to a complete standstill under peak loads. Congestion, then, is usually defined as a condition in the network where an increase in traffic load causes a decrease in throughput. The best point at which to operate the network is to the left of the dashed line in Figure 4.16, by controlling the offered load directly with, for instance, a rate control method. In some studies it was found that the optimal point is at the *knee* of the curve in Figure 4.16: the saturation point of the network under ideal conditions. Optimization is then interpreted as the maximization of throughput divided by measured round-trip message delay.

Rate control and flow control can be applied independently of one another. A standard rate control method is to give the sender a *permit* to offer data to the network at a specific average number of bytes per second. It can specify two parameters:

- The average data rate *R* in bytes per second
- The averaging interval that is used to calculate *R*

In the XTP protocol (see the Bibliographic Notes to Chapter 2) a third parameter is used:

- The maximum data burst rate

Rate control is important as an efficiency and network control issue. It cannot, however, affect the logical consistency of a protocol definition, which is the primary focus of this book.

4.6 SUMMARY

Problems such as the ones we have discussed in this chapter have been discovered in many real-life protocols, and protocol designers will continue to be confronted with them over and over again. We have presented them here in their most basic form, to identify where the potential design flaws are.

Flow control and error control are often hard to distinguish. A flow control scheme can be used to coordinate the rate of transmission of messages between the processes in a distributed system. It can be used to avoid bottlenecks, and to recover from transmission errors. The strategies we have explored include the use of timeouts, the extension of messages with sequence numbers, and the use of positive and negative acknowledgments. A logical extension of static window flow control mechanism is dynamic window flow control. It allows protocols to become self-adapting, a principle of sound design. Flow control methods can be used to solve a variety of problems. They can be used in an end-to-end protocol to synchronize a sender and a receiver. They can be used in link level protocols to optimize buffer management and bandwidth utilization. Finally, they can be used as specific congestion avoidance techniques to match the capacity of a sender to the capacity of the network that carries the traffic.

Throughout this chapter we have assumed that a receiver process can establish whether incoming messages should be acknowledged and accepted, or should be rejected due to transmission errors. Refer to Chapter 3 to see how this can be

accomplished.

EXERCISES

4-1. Describe in detail the conditions under which an *X-on/X-off* protocol and a *Ping-Pong* (stop-and-wait) protocol can fail. □

4-2. Consider the adequacy of the alternating bit protocol under message loss, duplication, and reordering. □

4-3. Change the extended alternating bit protocol from Figures 4.13 and 4.14 by also sending a negative acknowledgment when a message is received with the wrong sequence number. Show precisely what can go wrong. □

4-4. Extend the X-on/X-off protocol for full-duplex transmissions. Consider the extra problems that the loss of control messages can now cause. □

4-5. Show what happens if the timeout period in the alternating bit protocol is not chosen correctly. □

4-6. If the acknowledgment message in the alternating bit protocol is delayed long enough to trigger the sender's timeout, a duplicate *mesg* from the sender is created, which in turn triggers a duplicate *ack* message, and so on. How would you change the protocol to solve this problem? □

4-7. Describe your favorite traffic control problem (for example, grid lock, right of way problems, traffic circles) as a protocol problem. □

4-8. Two divisions of an army are encamped to the south and to the north of a guerrilla force that is slightly stronger than either of the two divisions separately. Together, however, the two divisions can launch a surprise attack and defeat their adversaries. The problem for them is to coordinate their plans such that neither will mistakenly attack alone. It is decided beforehand that division *A* will notify division *B* of the plan for attack by sending a messenger. The messenger, though, must pass guerrilla-held territory to reach his goal. This ''communication channel'' between *A* and *B* is expected to have a substantial loss rate, and at least a potential for message distortion and insertion. Assume that message distortion can be dealt with by using proper encoding techniques. There is a flow control problem caused by the disappearance and reappearance of detained messengers. It is decided that to confirm the safe arrival of a messenger from *A* to *B* a second messenger will be sent from *B* to *A* with an acknowledgment. But, when can division *B* be sure that its acknowledgment arrived? The acknowledgment has to survive the same channel behavior as the original message. Therefore, the acknowledgment must itself be acknowledged. But in that case, the acknowledgment of acknowledgments would have to continue *ad infinitum*. What is the flaw in this reasoning? (This is a ''folk'' problem in protocol theory; for instance, see Bertsekas and Gallager [1987, pp. 28-29.]). □

4-9. In a sliding window protocol where messages are not accepted out of order, show what can happen when the window size *W* equals to the range of the sequence numbers *M* (see Figure 4.11). □

4-10. Show how you can reduce the dimensions of all four arrays in the protocol of Figure 4.11 to the maximum window size. □

4-11. Consider the following problem on a channel that can reorder messages. A message with sequence number N is sent and acknowledged by the receiver, but the acknowledgment suffers a very long delay in the channel. A timeout occurs, and the message numbered N is retransmitted. The new acknowledgment overtakes the old one. The window of the sliding window protocol advances, and after it has advanced one full cycle, a new message with sequence number N is transmitted. By this time, the old acknowledgment finally makes it back to the sender and is confused for a new acknowledgment for the last message sent. Can you devise a solution to this problem? □

4-12. An alternative method for the calculation of a retransmission timeout, used in the TCP protocol, is based on the following formula Stallings [1985, p. 508], Zhang [1986], Karn and Partridge [1987]: $\beta \cdot (\alpha \cdot \overline{T} + (1-\alpha) \cdot T_{last})$, where T_{last} is the last observed round-trip delay. Compare this method with the one given in this chapter. Explain the effect of parameters α and β. □

4-13. The original alternating bit protocol, shown in Figure 4.8, is only partially specified. Provide the missing pieces. □

4-14. Consider in detail what might happen if, in Figure 4.12, *recvd[p]* would be reset to *false* in the accept process immediately after an acknowledgment is sent. □

BIBLIOGRAPHIC NOTES

The ‘‘alternating bit protocol,’’ introduced in this chapter, is one of the simplest, best documented, and most thoroughly verified protocol designs. It was first described in a paper by three people from the National Physical Laboratory in England, Bartlett, Scantlebury and Wilkinson [1969], in response to an article by W.C. Lynch [1968]. Variations of the NPL protocol are still popular as a litmus test for new protocol validation and specification methods. Cerf and Kahn [1974] first extended the alternating bit protocol into a go-back-N sliding window protocol. The selective repeat strategy is due to Stenning [1976]. The block acknowledgment strategy was first described in Brown, Gouda, and Miller [1989].

A general introduction to flow control techniques can be found in, for instance, Pouzin [1976], Tanenbaum [1981, 1988], or Stallings [1985]. An excellent survey and comparison of flow control techniques was published in Gerla and Kleinrock [1980]. An early attempt at rate control is described in Beeforth et al. [1972]. It distinguishes between two types of acknowledgment: one acknowledges to the sender that a message was correctly received and need not be retransmitted, and another signals to the sender that the buffer space occupied by that message was released (e.g., because the packet was forwarded), and that the window of sequence numbers can advance a notch.

Various versions of Figure 4.16 have been published over the years. It is discussed in detail in, for instance, Gerla and Kleinrock [1980] and Jain [1986].

The XTP, or Express Transfer Protocol is described in Chesson [1987]. The protocol was designed to survive applications in high speed data networks. It is promoted by the company ‘‘Protocol Engines,’’ founded by Greg Chesson. Other important work on protocols for high-speed data networks is reported in Clark [1985], and Clark,

Lambert and Zhang [1988]. Dynamic window flow control methods are described in, for instance, Gerla and Kleinrock [1980], Jain [1986]. Jacobson's slow start protocol is described in Jacobson [1988].

More on the choice of timeout intervals for network protocols can be found in Zhang [1986] and Karn and Partridge [1987]. For an introduction to general network control issues refer to McQuillan and Walden [1977], Tanenbaum [1981, 1988], Cole [1987], or Stallings [1985, 1988].