Online Book

3.4: Principles of Reliable Data Transfer

In this section, we consider the problem of reliable data transfer in a general context. This is appropriate since the problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well. The general problem is thus of central importance to networking. Indeed, if one had to identify a "top-10" list of fundamentally important problems in all of networking, this would be a top candidate to lead that list. In the next section we will examine TCP and show, in particular, that TCP exploits many of the principles that we are about to describe.

Figure 3.8 illustrates the framework for our study of reliable data transfer. The service abstraction provided to the upper layer entities is that of a reliable channel through which data can be transferred. With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.

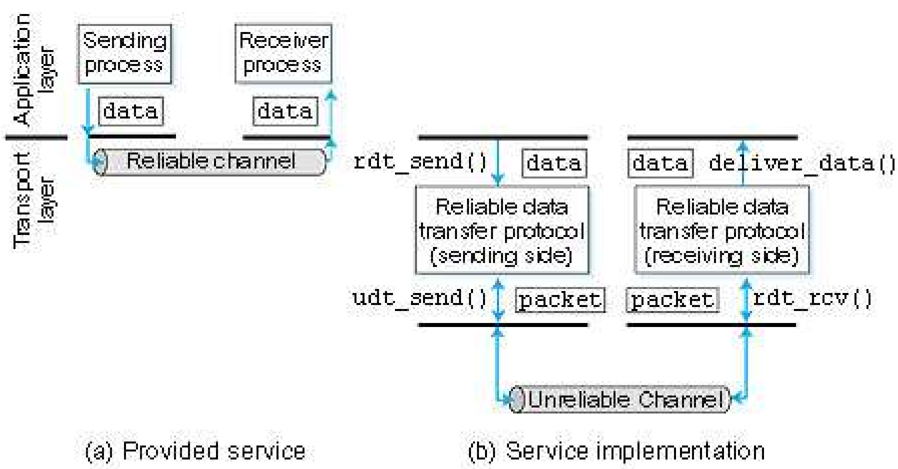

Figure 3.8: Reliable data transfer: Service model and service implementation

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that the layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-end network layer. More generally, the layer beneath the two reliably communicating endpoints might consist of a single physical link (for example, as in the case of a link-level data transfer protocol) or a global internetwork (for example, as in the case of a transport-level protocol). For our purposes, however, we can view this lower layer simply as an unreliable point-to-point channel.

In this section, we will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel. Figure 3.8(b) illustrates the interfaces for our data transfer protocol. The sending side of the data transfer protocol will be invoked from above by a call to `rdt_send()`. It will be passed the data to be delivered to the upper layer at the receiving side. (Here `rdt` stands for "reliable data transfer" protocol and `_send` indicates that the sending side of `rdt` is being called. The first step in developing any protocol is to choose a good name!) On the receiving side, `rdt_rcv()` will be called when a packet arrives from the receiving side of the channel. When the `rdt` protocol wants to deliver data to the upper layer, it will do so by calling `deliver_data()`. In the following we use the terminology "packet" rather than "segment" for the protocol data unit. Because the theory developed in this section applies to computer networks in general and not just to the Internet transport layer, the generic term "packet" is perhaps more appropriate here.

In this section we consider only the case of **unidirectional** data transfer, that is, data transfer from the sending to receiving side. The case of reliable **bidirectional** (that is, full duplex) data transfer is conceptually no more difficult but considerably more tedious to explain. Although we consider only unidirectional data transfer, it is important to note that the sending and receiving sides of our protocol will nonetheless need to transmit packets in *both* directions, as indicated in Figure 3.8. We will see shortly that, in addition to exchanging packets containing the data to be transferred, the sending and receiving sides of `rdt` will also need to exchange control packets back and forth. Both the send and receive sides of `rdt` send packets to the other side by a call to `udt_send()` (where `udt` stands for unreliable data transfer).

3.4.1: Building a Reliable Data-Transfer Protocol

We now step through a series of protocols, each one becoming more complex, arriving at a flawless reliable data transfer protocol.

Reliable Data Transfer Over a Perfectly Reliable Channel: `rdt1.0`

We first consider the simplest case in which the underlying channel is completely reliable. The protocol itself, which we'll call `rdt1.0`, is trivial. The **finite-state machine (FSM)** definitions for the `rdt1.0` sender and receiver are shown in Figure 3.9. The sender and receiver FSMs in Figure 3.9 each have just one state. The arrows in the FSM description indicate the transition of the protocol from one state to another. (Since each FSM in Figure 3.9 has just one state, a transition is necessarily from the one state back to itself; we'll see more complicated state diagrams shortly.) The event causing the transition is shown above the horizontal line labeling the transition, and the action(s) taken when the event occurs are shown below the horizontal line.

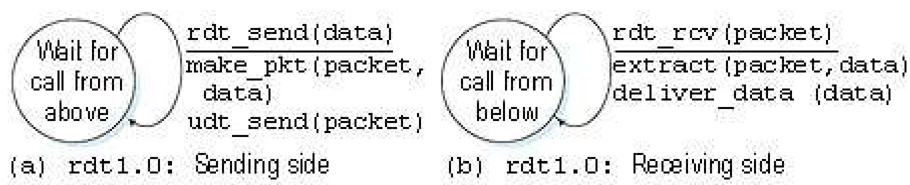

Figure 3.9: `rdt1.0`--A protocol for a completely reliable channel

The sending side of `rdt` simply accepts data from the upper layer via the `rdt_send(data)` event, puts the data into a packet (via the action `make_pkt (packet,data)`) and sends the packet into the channel. In practice, the `rdt_send(data)` event would result from a procedure call (for example, to `rdt_send()`) by the upper-layer application.

On the receiving side, `rdt` receives a packet from the underlying channel via the `rdt_rcv(packet)` event, removes the data from the packet (via the action `extract(packet,data)`) and passes the data up to the upper layer. In practice, the `rdt_rcv(packet)` event would result from a procedure call (for example, to `rdt_rcv()`) from the lower-layer protocol.

In this simple protocol, there is no difference between a unit of data and a packet. Also, all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong! Note that we have also assumed that the receiver is able to receive data as fast as the sender happens to send data. Thus, there is no need for the receiver to ask the sender to "slow down!"

Reliable Data Transfer Over a Channel with Bit Errors: `rdt2.0`

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted. Such bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered. We'll continue to assume for the moment that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

Before developing a protocol for reliably communicating over such a channel, first consider how people might deal with such a situation. Consider how you yourself might dictate a long message over the phone. In a typical scenario, the message taker might say "OK" after each sentence has been heard, understood, and recorded. If the message taker hears a garbled sentence, you're asked to repeat the garbled sentence. This message-dictation protocol uses both **positive**

acknowledgments ("OK") and **negative acknowledgments** ("Please repeat that."). These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating. In a computer network setting, reliable data transfer protocols based on such retransmission are known **ARQ** (Automatic Repeat reQuest) **protocols.**

Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

- *Error detection.* First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. Recall from the previous section that UDP uses the Internet checksum field for exactly this purpose. In Chapter 5 we'll examine error detection and correction techniques in greater detail; these techniques allow the receiver to detect and possibly correct packet bit errors. For now, we need only know that these techniques require that extra bits (beyond the bits of original data to be transferred) be sent from the sender to receiver; these bits will be gathered into the packet checksum field of the `rdt2.0` data packet.

- *Receiver feedback.* Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world (in this case, whether or not a packet was received correctly) is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message dictation scenario are examples of such feedback. Our `rdt2.0` protocol will similarly send ACK and NAK packets back from the receiver to the sender. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

- *Retransmission.* A packet that is received in error at the receiver will be retransmitted by the sender.

Figure 3.10 shows the FSM representation of `rdt2.0`, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

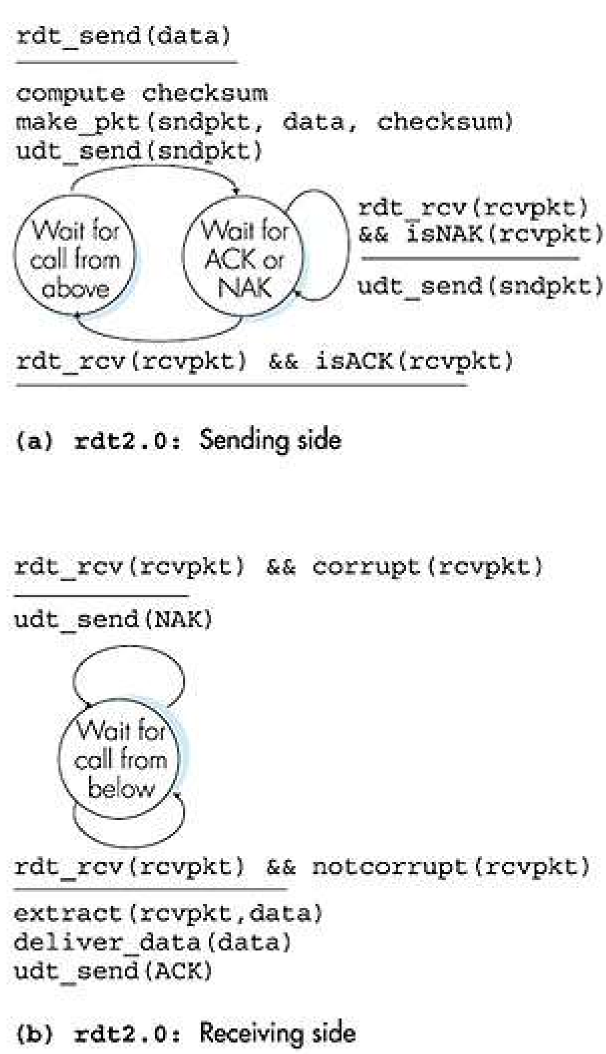

Figure 3.10: `rdt2.0`--A protocol for a channel with bit errors

The send side of `rdt2.0` has two states. In one state, the send-side protocol is waiting for data to be passed down from the upper layer. In the other state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation `rdt_rcv(rcvpkt) && isACK (rcvpkt)` in Figure 3.10 corresponds to this event), the sender knows the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer. If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet. It is important to note that when the receiver is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that will only happen after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as `rdt2.0` are known as **stop-and-wait** protocols.

The receiver-side FSM for `rdt2.0` still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted. In Figure 3.10, the notation `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` corresponds to the event in which a packet is received and is found to be in error.

Protocol `rdt2.0` may look as if it works but, unfortunately, it has a fatal flaw. In particular, we haven't accounted for the possibility that the ACK or NAK packet could be corrupted! (Before proceeding on, you should think about how this problem may be fixed.) Unfortunately, our slight oversight is not as innocuous as it may seem. Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors. The more difficult question is how the protocol should recover from errors in ACK or NAK packets. The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data.

Consider three possibilities for handling corrupted ACKs or NAKs:

- For the first possibility, consider what a human might do in the message dictation scenario. If the speaker didn't understand the "OK" or "Please repeat that" reply from the receiver, the speaker would probably ask "What did you say?" (thus introducing a new type of sender-to-receiver packet to our protocol). The speaker would then repeat the reply. But what if the speaker's "What did you say?" is corrupted? The receiver, having no idea whether the garbled sentence was part of the dictation or a request to repeat the last reply, would probably then respond with "What did *you* say?" And then, of course, that response might be garbled. Clearly, we're heading down a difficult path.

- A second alternative is to add enough checksum bits to allow the sender to not only detect, but to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.

- A third approach is for the sender to simply resend the current data packet when it receives a garbled ACK or NAK packet. This method, however, introduces **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know a priori whether an arriving packet contains new data or is a retransmission!

A simple solution to this new problem (and one adopted in almost all existing data-transfer protocols including TCP) is to add a new field to the data packet and have the sender number its data packets by putting a **sequence number** into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission. For this simple case of a stop-and-wait protocol, a one-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet (the sequence number changes, moving "forward" in modulo-2 arithmetic). Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging. The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

Figures 3.11 and 3.12 show the FSM description for `rdt2.1`, our fixed version of `rdt2.0`. The `rdt2.1` sender and receiver FSM's each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1. Note that the actions in those states where a 0-numbered packet is being sent or expected are mirror images of those where a 1-numbered packet is being sent or expected; the only differences have to do with the handling of the sequence number.

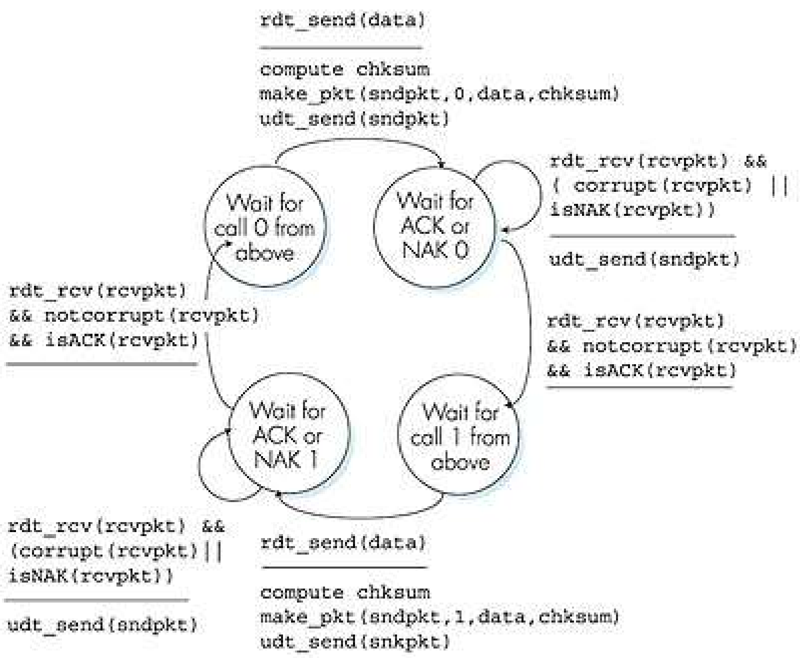

Figure 3.11: `rdt2.1` sender

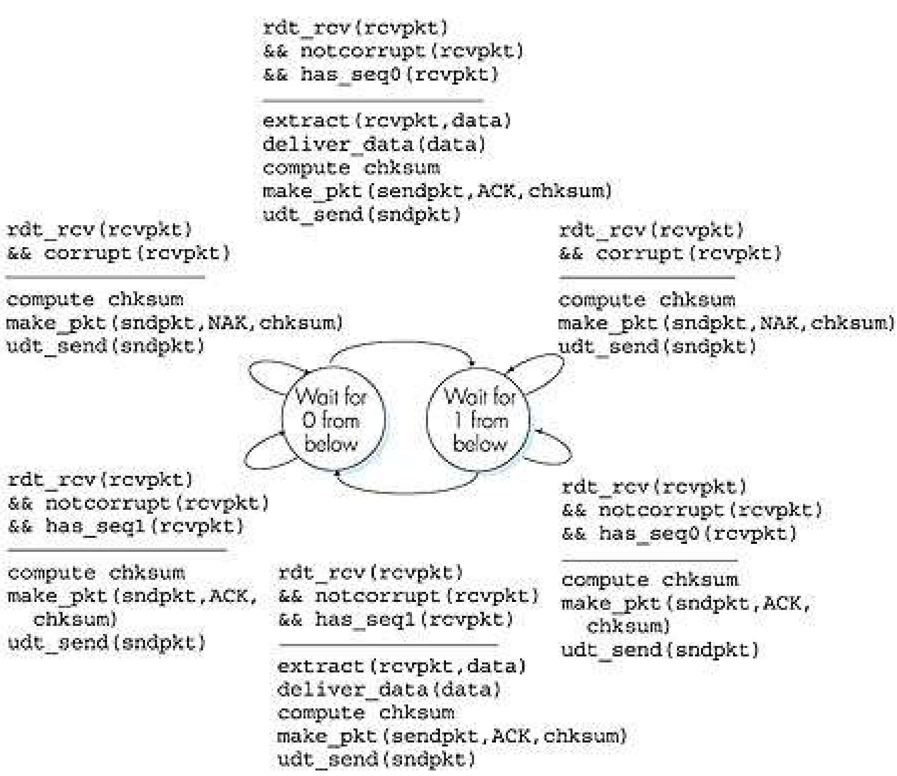

Figure 3.12: `rdt2.1` receiver

Protocol `rdt2.1` uses both positive and negative acknowledgments from the receiver to the sender. When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received. When a corrupted packet is received, the receiver sends a negative acknowledgment. We can accomplish the same effect as a NAK if, instead of sending a NAK, we instead send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet (that is, receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Many TCP implementations use the receipt of so-called "triple duplicate ACKs" (three ACK packets all acknowledging the same already-acknowledged packet) to trigger a retransmission at the sender. Our NAK-free reliable data transfer protocol for a channel with bit errors is `rdt2.2`, shown in Figures 3.13 and 3.14.

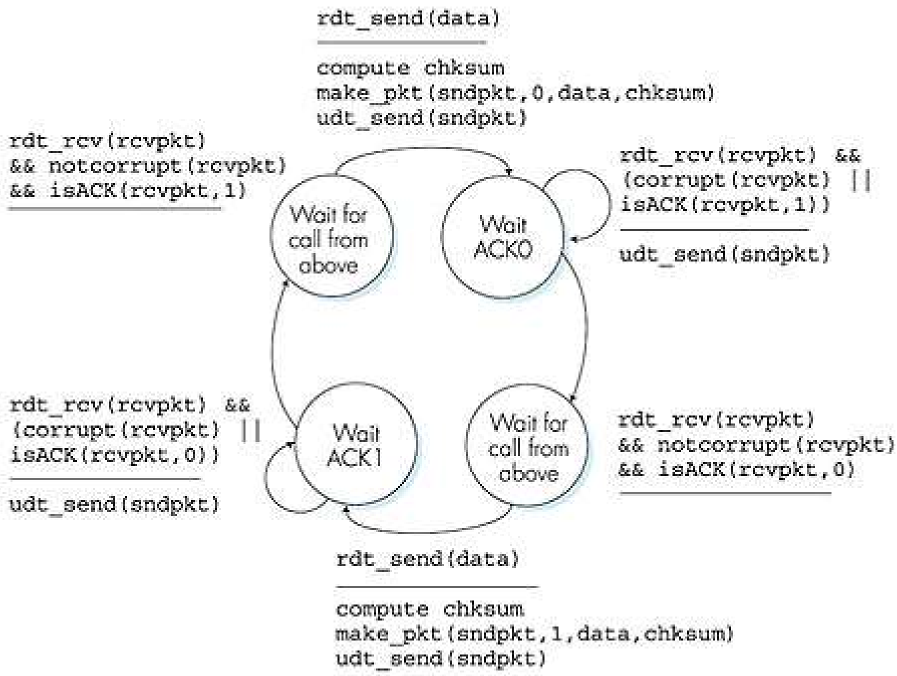

Figure 3.13: `rdt2.2` sender

Figure 3.14: `rdt2.2` receiver

Reliable Data Transfer Over a Lossy Channel with Bit Errors: `rdt3.0`

Suppose now that in addition to corrupting bits, the underlying channel can *lose* packets as well, a not-uncommon event in today's computer networks (including the Internet). Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions--the techniques already developed in `rdt2.2`--will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism.

There are many possible approaches toward dealing with packet loss (several more of which are explored in the exercises at the end of the chapter). Here, we'll put the burden of detecting and recovering from lost packets on the sender. Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is *certain* that a packet has been lost, it can simply retransmit the data packet. You should convince yourself that this protocol does indeed work.

But how long must the sender wait to be certain that something has been lost? The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers or gateways) plus whatever amount of time is needed to process a packet at the receiver. In many networks, this worst-case maximum delay is very difficult to even estimate, much less know with certainty. Moreover, the protocol should ideally recover from packet loss as soon as possible; waiting for a worst-case delay could mean a long wait until error recovery is initiated. The approach thus adopted in practice is for the sender to "judiciously" choose a time value such that packet loss is likely, although not guaranteed, to have happened. If an ACK is not received within this time, the packet is retransmitted. Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost. This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Happily, protocol `rdt2.2` already has enough functionality (that is, sequence numbers) to handle the case of duplicate packets.

From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. In all cases, the action is the same: retransmit. In order to implement a time-based retransmission mechanism, a **countdown timer** will be needed that can interrupt the sender after a given amount of timer has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet, or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

The existence of sender-generated duplicate packets and packet (data, ACK) loss also complicates the sender's processing of any ACK packet it receives. If an ACK is received, how is the sender to know if it was sent by the receiver in response to its (sender's) own most recently transmitted packet, or is a delayed ACK sent in response to an earlier transmission of a different data packet? The solution to this dilemma is to augment the ACK packet with an **acknowledgment field.** When the receiver generates an ACK, it will copy the sequence number of the data packet being acknowledged into this acknowledgment field. By examining the contents of the acknowledgment field, the sender can determine the sequence number of the packet being positively acknowledged.

Figure 3.15 shows the sender FSM for `rdt3.0`, a protocol that reliably transfers data over a channel that can corrupt or lose packets. Figure 3.16 shows how the protocol operates with no lost or delayed packets, and how it handles lost data packets.

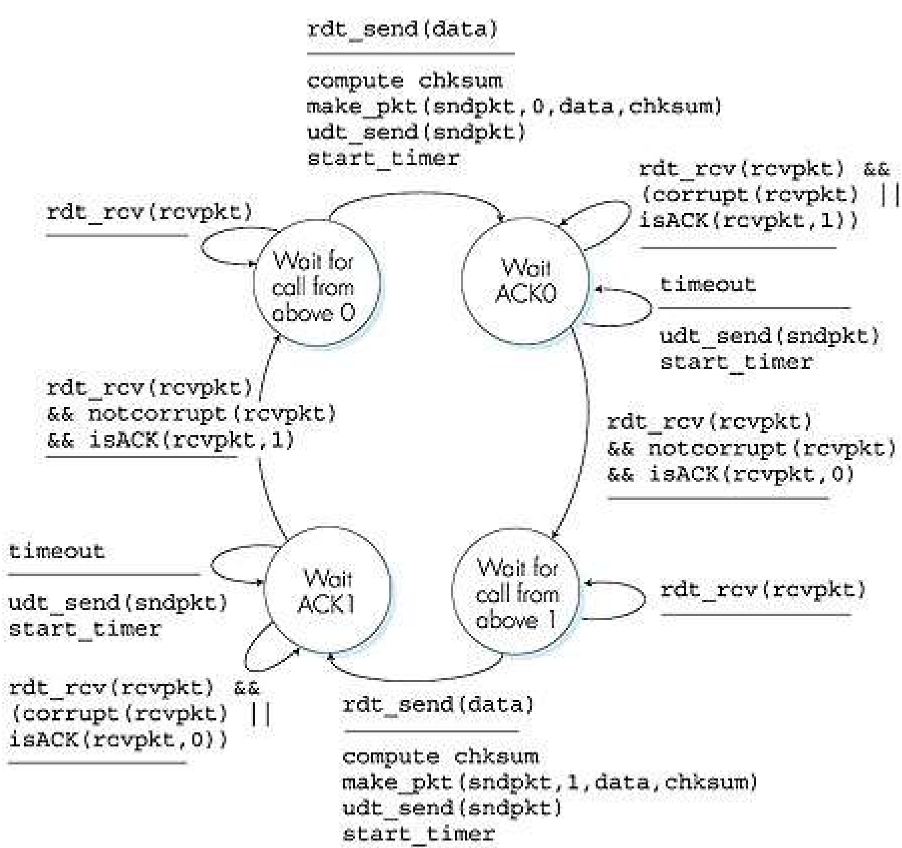

Figure 3.15: `rdt3.0` sender FSM

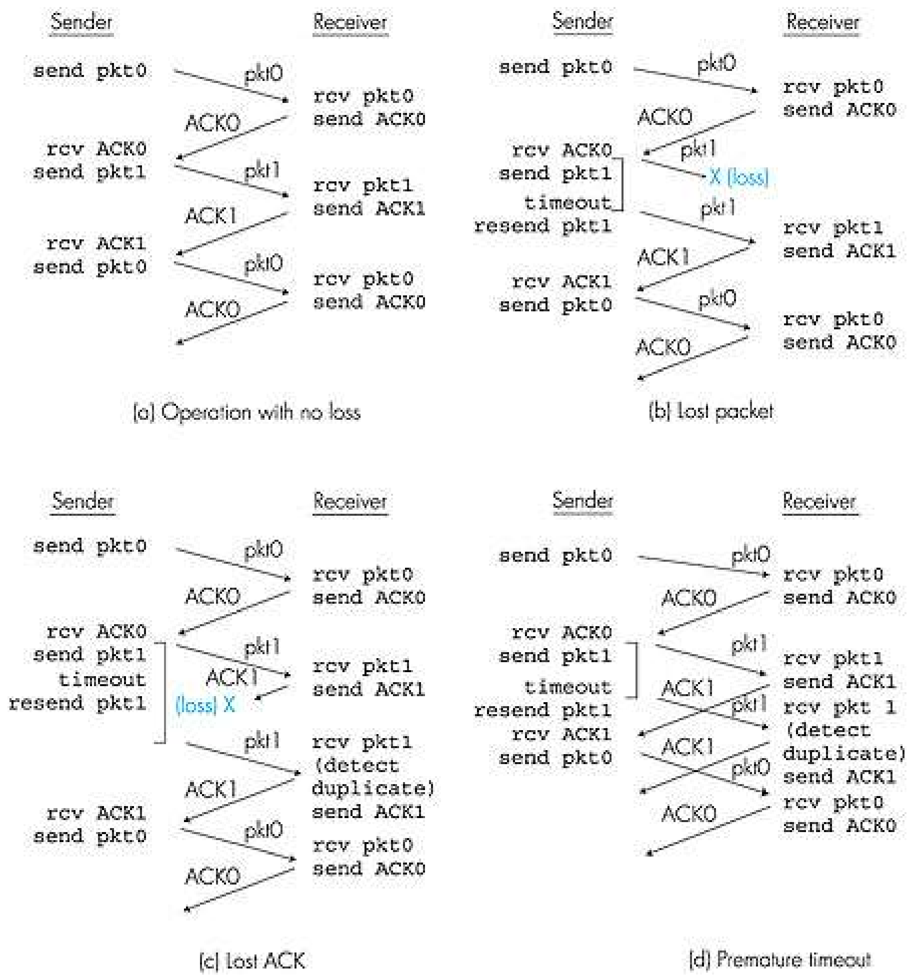

Figure 3.16: Operation of `rdt3.0`, the alternating-bit protocol

In Figure 3.16, time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. In Figures 3.16b-d, the send-side brackets indicate the times at which a timer is set and later times out. Several of the more subtle aspects of this protocol are explored in the exercises at the end of this chapter. Because packet sequence numbers alternate between 0 and 1, protocol `rdt3.0` is sometimes known as the **alternating-bit protocol.** We have now assembled the key elements of a data-transfer protocol. Checksums, sequence numbers, timers, and positive and negative acknowledgment packets each play a crucial and necessary role in the operation of the protocol. We now have a working reliable data-transfer protocol!

3.4.2: Pipelined Reliable Data Transfer Protocols

Protocol `rdt3.0` is a functionally correct protocol, but it is unlikely that anyone would be happy with its performance, particularly in today's high-speed networks. At the heart of `rdt3.0`'s performance problem is the fact that it is a stop-and-wait protocol.

To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two end hosts, one located on the West Coast of the United States and the other located on the East Coast. The speed-of-light propagation delay, T_{prop}, between these two end systems is approximately 15 milliseconds. Suppose that they are connected by a channel with a capacity, C, of 1 gigabit (10^9 bits) per second. With a packet size, SP, of 1 Kbytes per packet including both header fields and data, the time needed to actually transmit the packet into the 1Gbps link is

$$T_{\text{trans}} = \frac{SP}{C} = \frac{8\,\text{KBits/packet}}{10^9\,\text{bits/sec}} = 8\,\text{microseconds}$$

With our stop-and-wait protocol, if the sender begins sending the packet at t = 0, then at t = 8 microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, as depicted in Figure 3.17a, with the last bit of the packet emerging at the receiver at t = 15.008 msec.

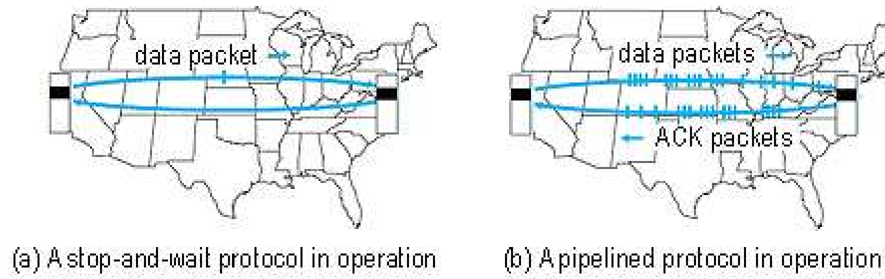

Figure 3.17: Stop-and-wait versus pipelined protocol

Assuming for simplicity that ACK packets are the same size as data packets and that the receiver can begin sending an ACK packet as soon as the last bit of a data packet is received, the last bit of the ACK packet emerges back at the sender at *t* = 30.016 msec. Thus, in 30.016 msec, the sender was busy (sending or receiving) for only 0.016 msec. If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits via the channel, we have a rather dismal sender utilization, U_{sender}, of

$$U_{\text{sender}} = \frac{.008}{30.016} = 0.00027$$

That is, the sender was busy only 2.7 hundredths of one percent of the time. Viewed another way, the sender was only able to send 1 kilobyte in 30.016 milliseconds, an effective throughput of only 33 kilobytes per second--even though a 1 gigabit per second link was available! Imagine the unhappy network manager who just paid a fortune for a gigabit capacity link but manages to get a throughput of only 33 kilobytes per second! This is a graphic example of how network protocols can limit the capabilities provided by the underlying network hardware. Also, we have neglected lower-layer protocol-processing times at the sender and receiver, as well as the processing and queuing delays that would occur at any intermediate routers between the sender and receiver. Including these effects would only serve to further increase the delay and further accentuate the poor performance.

The solution to this particular performance problem is a simple one: rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as shown in Figure 3.17(b). Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining.** Pipelining has several consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.
- The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted, but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.

The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: **Go-Back-N** and **selective repeat.**

3.4.3: Go-Back-N (GBN)

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline. Figure 3.18 shows the sender's view of the range of sequence numbers in a GBN protocol. If we define `base` to be the sequence number of the oldest unacknowledged packet and `nextseqnum` to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval `[0,base-1]` correspond to packets that have already been transmitted and acknowledged. The interval `[base,nextseqnum-1]` corresponds to packets that have been sent but not yet acknowledged. Sequence numbers in the interval `[nextseqnum,base+N-1]` can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to `base+N` cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged.

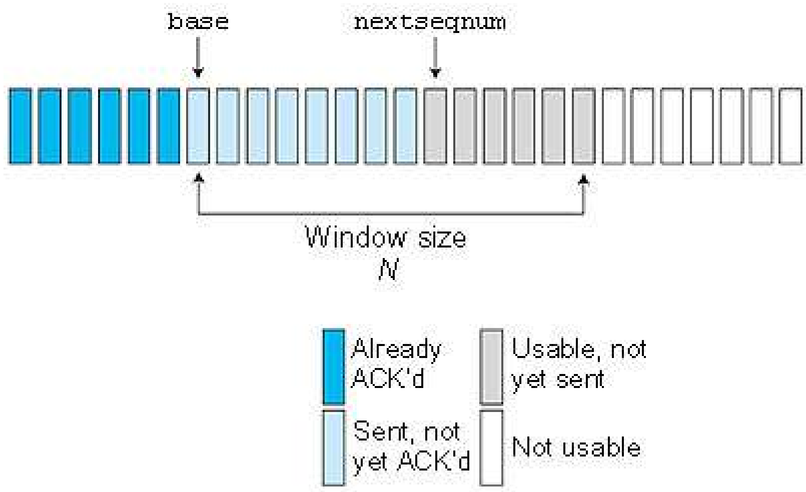

Figure 3.18: Sender's view of sequence numbers in Go-Back-N

As suggested by Figure 3.18, the range of permissible sequence numbers for transmitted but not-yet-acknowledged packets can be viewed as a "window" of size *N* over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol.** You might be wondering why we would even limit the number of outstanding, unacknowledged packets to a value of *N* in the first place. Why not allow an unlimited number of such packets? We will see in Section 3.5 that flow control is one reason to impose a limit on the sender. We'll examine another reason to do so in Section 3.7, when we study TCP congestion control.

In practice, a packet's sequence number is carried in a fixed-length field in the packet header. If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0,2^k - 1]$. With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2^k arithmetic. (That is, the sequence number space can be thought of as a ring of size 2^k, where sequence number $2^k - 1$ is immediately followed by sequence number 0.) Recall that `rdt3.0` had a one-bit sequence number and a range of sequence numbers of [0,1]. Several of the problems at the end of this chapter explore consequences of a finite range of sequence numbers. We will see in Section 3.5 that TCP has a 32-bit sequence-number field, where TCP sequence numbers count bytes in the byte stream rather than packets.

Figures 3.19 and 3.20 give an extended-FSM description of the sender and receiver sides of an ACK-based, NAK-free, GBN protocol. We refer to this FSM description as an **extended FSM** since we have added variables (similar to programming language variables) for `base` and `nextseqnum` and also added operations on these variables and conditional actions involving these variables. Note that the extended FSM specification is now beginning to look somewhat like a programming language specification. [Bochman 1984] provides an excellent survey of additional extensions to FSM techniques as well as other programming language-based techniques for specifying protocols.

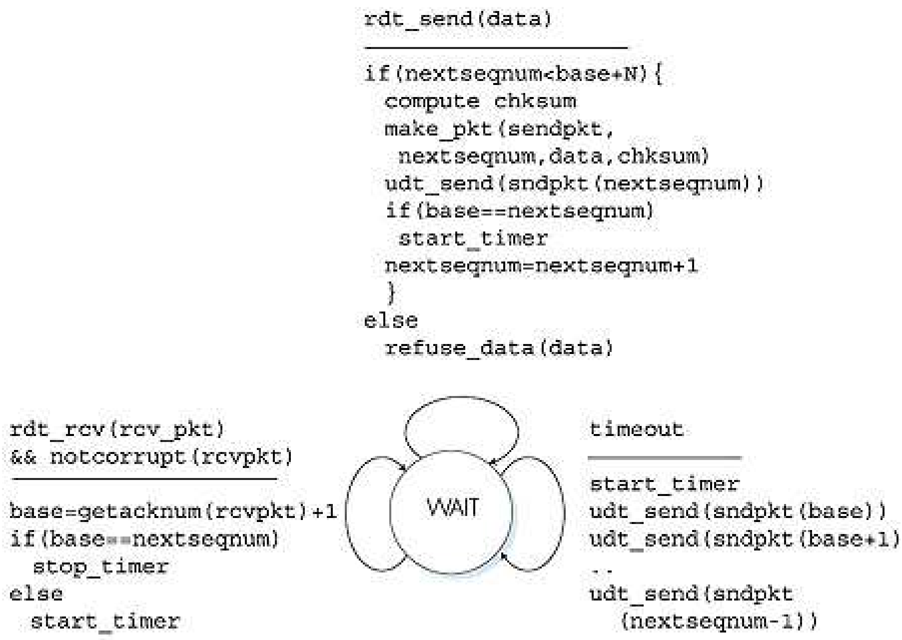

Figure 3.19: Extended FSM description of GBN sender

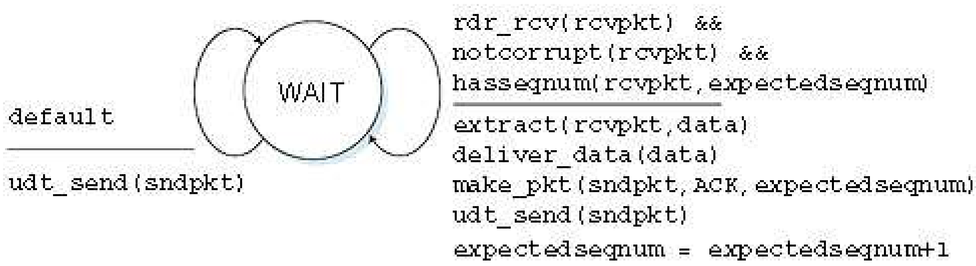

Figure 3.20: Extended FSM description of GBN receiver

The GBN sender must respond to three types of events:

- *Invocation from above.* When `rdt_send()` is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call `rdt_send()` only when the window is not full.

- *Receipt of an ACK.* In our GBN protocol, an acknowledgment for packet with sequence number *n* will be taken to be a **cumulative acknowledgment,** indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver. We'll come back to this issue shortly when we examine the receiver side of GBN.

- *A timeout event.* The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. Our sender in Figure 3.19 uses only a single timer, which can be thought of as a timer for the oldest transmitted-but-not-yet-acknowledged packet. If an ACK is received but there are still additional transmitted-but-yet-to-be-acknowledged packets, the timer is restarted. If there are no outstanding unacknowledged packets, the timer is stopped.

The receiver's actions in GBN are also simple. If a packet with sequence number *n* is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number *n* - 1), the receiver sends an ACK for packet *n* and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. Note that since packets are delivered one-at-a-time to the upper layer, if packet *k* has been received and delivered, then all packets with a sequence number lower than *k* have also been delivered. Thus, the use of cumulative acknowledgments is a natural choice for GBN.

In our GBN protocol, the receiver discards out-of-order packets. Although it may seem silly and wasteful to discard a correctly received (but out-of-order) packet, there is some justification for doing so. Recall that the receiver must deliver data, in order, to the upper layer. Suppose now that packet *n* is expected, but packet *n* + 1 arrives. Since data must be delivered in order, the receiver *could* buffer (save) packet *n* + 1 and then deliver this packet to the upper layer after it had later received and delivered packet *n*. However, if packet *n* is lost, both it and packet *n* + 1 will eventually be retransmitted as a result of the GBN retransmission rule at the sender. Thus, the receiver can simply discard packet *n* + 1. The advantage of this approach is the simplicity of receiver buffering--the receiver need not buffer *any* out-of-order packets. Thus, while the sender must maintain the upper and lower bounds of its window and the position of `nextseqnum` within this window, the only piece of information the receiver need maintain is the sequence number of the next in-order packet. This value is held in the variable `expectedseqnum`, shown in the receiver FSM in Figure 3.20. Of course, the disadvantage of throwing away a correctly received packet is that the subsequent retransmission of that packet might be lost or garbled and thus even more retransmissions would be required.

Figure 3.21 shows the operation of the GBN protocol for the case of a window size of four packets. Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding. As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively). On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out-of-order and are discarded.

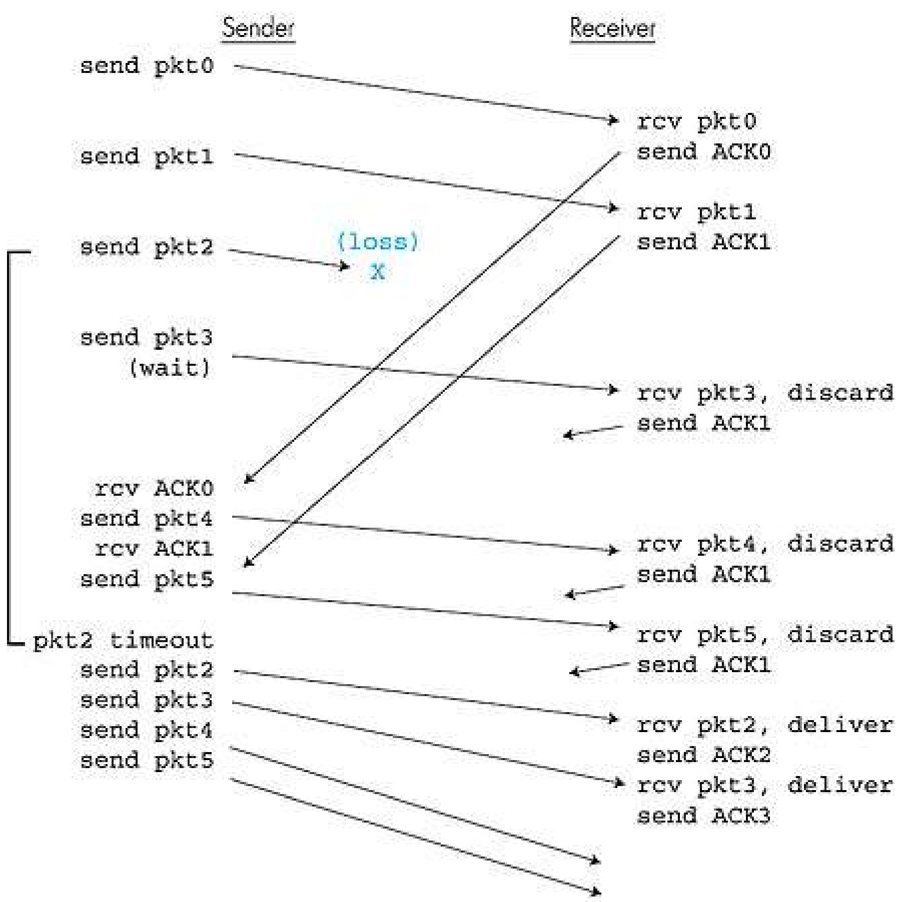

Figure 3.21: Go-Back-N in operation

Before closing our discussion of GBN, it is worth noting that an implementation of this protocol in a protocol stack would likely be structured similar to that of the extended FSM in Figure 3.19. The implementation would also likely be in the form of various procedures that implement the actions to be taken in response to the various events that can occur. In such **event-based programming,** the various procedures are called (invoked) either by other procedures in the protocol stack, or as the result of an interrupt. In the sender, these events would be (1) a call from the upper-layer entity to invoke `rdt_send()`, (2) a timer interrupt, and (3) a call from the lower layer to invoke `rdt_rcv()` when a packet arrives. The programming exercises at the end of this chapter will give you a chance to actually implement these routines in a simulated, but realistic, network setting.

We note here that the GBN protocol incorporates almost all of the techniques that we will encounter when we study the reliable data-transfer components of TCP in Section 3.5. These techniques include the use of sequence numbers, cumulative acknowledgments, checksums, and a time-out/retransmit operation. In this sense, TCP has a number of elements of a GBN-style protocol. There are, however, some differences between GBN and TCP. Many TCP implementations will buffer correctly received but out-of-order segments [Stevens 1994]. A proposed modification to TCP, the so-called selective acknowledgment [RFC 2581], will also allow a TCP receiver to selectively acknowledge a single out-of-order packet rather than cumulatively acknowledge the last correctly received packet. The notion of a selective acknowledgment is at the heart of the second broad class of pipelined protocols: the so-called selective-repeat protocols that we study below. TCP is thus probably best categorized as a hybrid of Go-Back-N and selective-repeat protocols.

3.4.4: Selective Repeat (SR)

The GBN protocol allows the sender to potentially "fill the pipeline" in Figure 3.17 with packets, thus avoiding the channel utilization problems we noted with stop-and-wait protocols. There are, however, scenarios in which GBN itself will suffer from performance problems. In particular, when the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many of which may be unnecessary. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions. Imagine in our message dictation scenario, if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated. The dictation would be slowed by all of the reiterated words.

As the name suggests, Selective-Repeat (SR) protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as-needed, retransmission will require that the receiver *individually* acknowledge correctly received packets. A window size of *N* will again be used to limit the number of outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the sender will have already received ACKs for some of the packets in the window. Figure 3.22 shows the SR sender's view of the sequence number space. Figure 3.23 details the various actions taken by the SR sender.

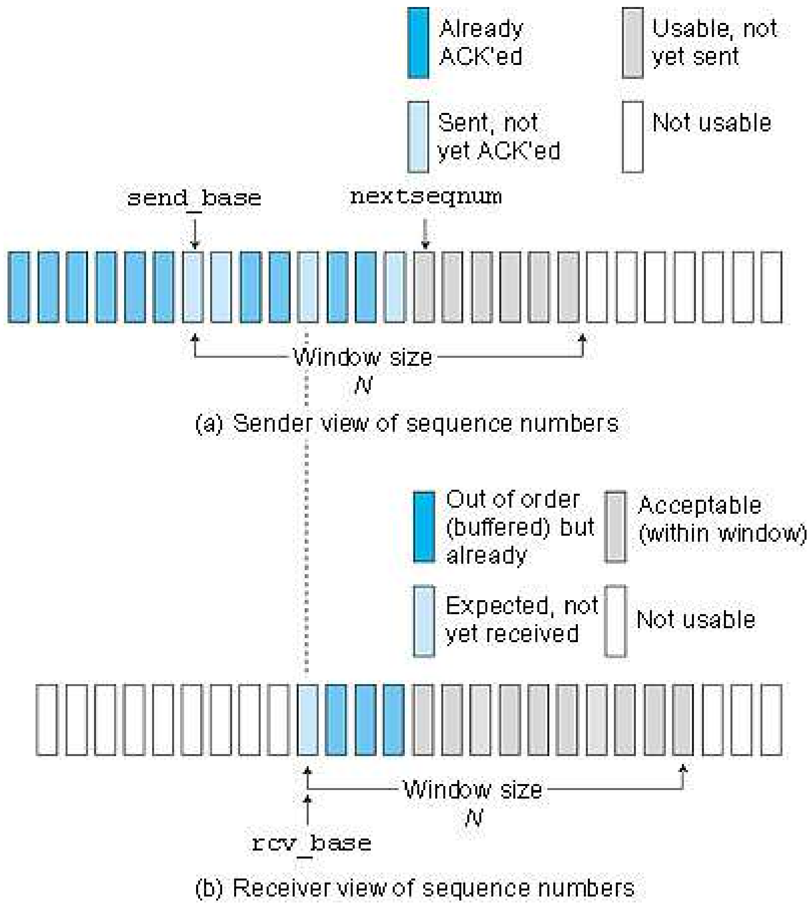

Figure 3.22: Selective Repeat (SR) sender and receiver views of sequence-number space

1. *Data received from above.* When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.

2. *Timeout.* Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers [Varghese 1997].

3. *ACK received.* If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to `send-base`, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

Figure 3.23: Selective Repeat (SR) sender events and actions

The SR receiver will acknowledge a correctly received packet whether or not it is in-order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in-order to the upper layer. Figure 3.24 itemizes the various actions taken by the SR receiver. Figure 3.25 shows an example of SR operation in the presence of lost packets. Note that in Figure 3.25, the receiver initially buffers packets 3 and 4, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

1. *Packet with sequence number in* `[rcv_base, rcv_base+N-1]` *is correctly received.* In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (`rcv_base` in Figure 3.22), then this packet, and any previously buffered and consecutively numbered (beginning with `rcv_base`) packets are delivered to the upper layer. The receive window is then moved forward by the number of

packets delivered to the upper layer. As an example, consider Figure 3.25. When a packet with a sequence number of `rcv_base=2` is received, it and packets `rcv_base+1 and rcv_base+2` can be delivered to the upper layer.

2. *Packet with sequence number in* `[rcv_base-N, rcv_base-1]` *is received.* In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.

3. *Otherwise.* Ignore the packet.

Figure 3.24: Selective Repeat (SR) receiver events and actions

Figure 3.25: SR operation (click to see in new window)

It is important to note that in step 2 in Figure 3.24, the receiver re-acknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base. You should convince yourself that this re-acknowledgment is indeed needed. Given the sender and receiver sequence-number spaces in Figure 3.22 for example, if there is no ACK for packet `send_base` propagating from the receiver to the sender, the sender will eventually retransmit packet `send_base`, even though it is clear (to us, not the sender!) that the receiver has already received that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward! This example illustrates an important aspect of SR protocols (and many other protocols as well). The sender and receiver will not always have an identical view of what has been received correctly and what has not. For SR protocols, this means that the sender and receiver windows will not always coincide.

The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3 and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios. In the first scenario, shown in Figure 3.26a, the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0--a copy of the first packet sent.

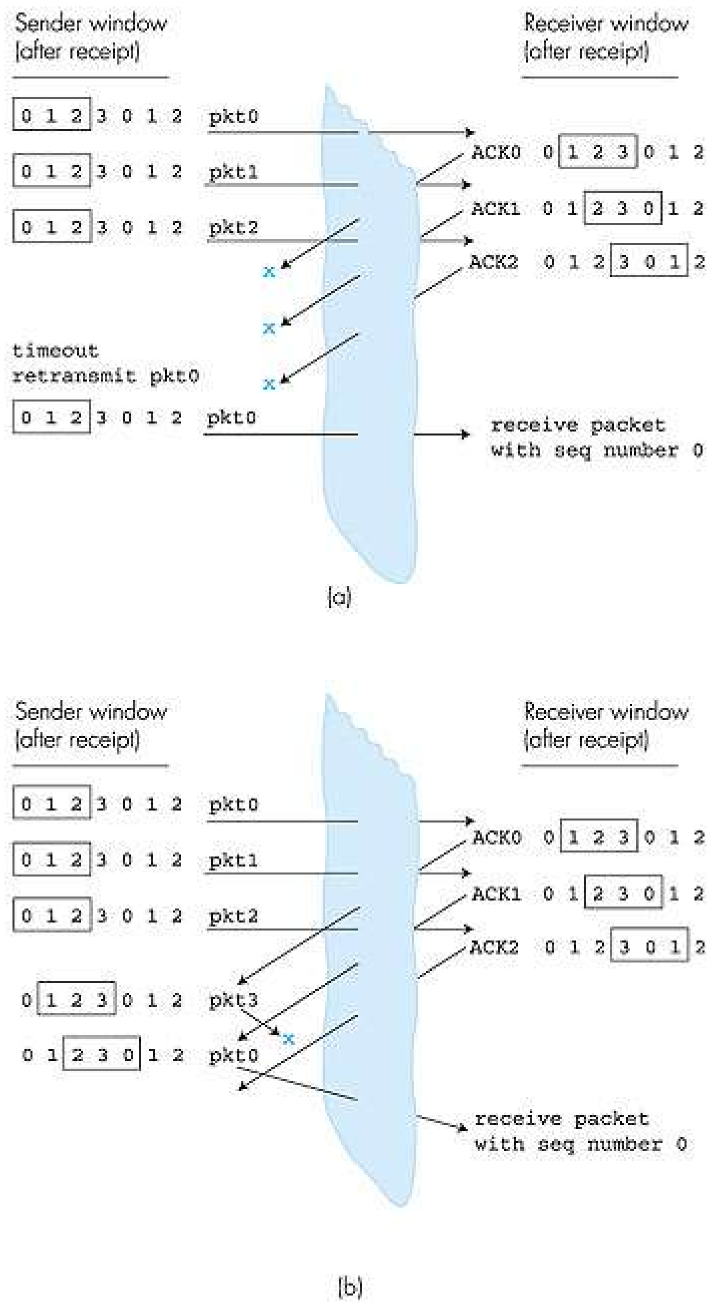

Figure 3.26: SR receiver dilemma with too-large windows: A new packet or retransmission?

In the second scenario, shown in Figure 3.26b, the ACKs for the first three packets are all delivered correctly. The sender thus moves its window forward and sends the fourth, fifth, and sixth packets, with sequence numbers 3, 0, 1, respectively. The packet with sequence number 3 is lost, but the packet with sequence number 0 arrives--a packet containing *new* data.

Now consider the receiver's viewpoint in Figure 3.26, which has a figurative curtain between the sender and the receiver, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the sequence of messages it receives from the channel and sends into the channel. As far as it is concerned, the two scenarios in Figure 3.26 are *identical.* There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet. Clearly, a window size that is 1 less than the size of the sequence number space won't work. But how small must the window size be? A problem at the end of the chapter asks you to show that the window size must be less than or equal to half the size of the sequence-number space for SR protocols.

Let us conclude our discussion of reliable data transfer protocols by considering one remaining assumption in our underlying channel model. Recall that we have assumed that packets cannot be reordered within the channel between the sender and receiver. This is generally a reasonable assumption when the sender and receiver are connected by a single physical wire. However, when the "channel" connecting the two is a network, packet reordering can occur. One manifestation of packet ordering is that old copies of a packet with a sequence or acknowledgment number of *x* can appear, even though neither the sender's nor the receiver's window contains *x*. With packet reordering, the channel can be thought of as essentially buffering packets and spontaneously emitting these packets at *any* point in the future. Because sequence numbers may be reused, some care must be taken to guard against such duplicate packets. The approach taken in practice is to ensure that a sequence number is not reused until the sender is relatively "sure" than any previously sent packets with sequence number *x* are no longer in the network. This is done by assuming that a packet cannot "live" in the network for longer than some fixed maximum amount of time. A maximum packet lifetime of approximately three minutes is assumed in the TCP extensions for high-speed networks [RFC 1323]. [Sunshine 1978] describes a method for using sequence numbers such that reordering problems can be completely avoided.