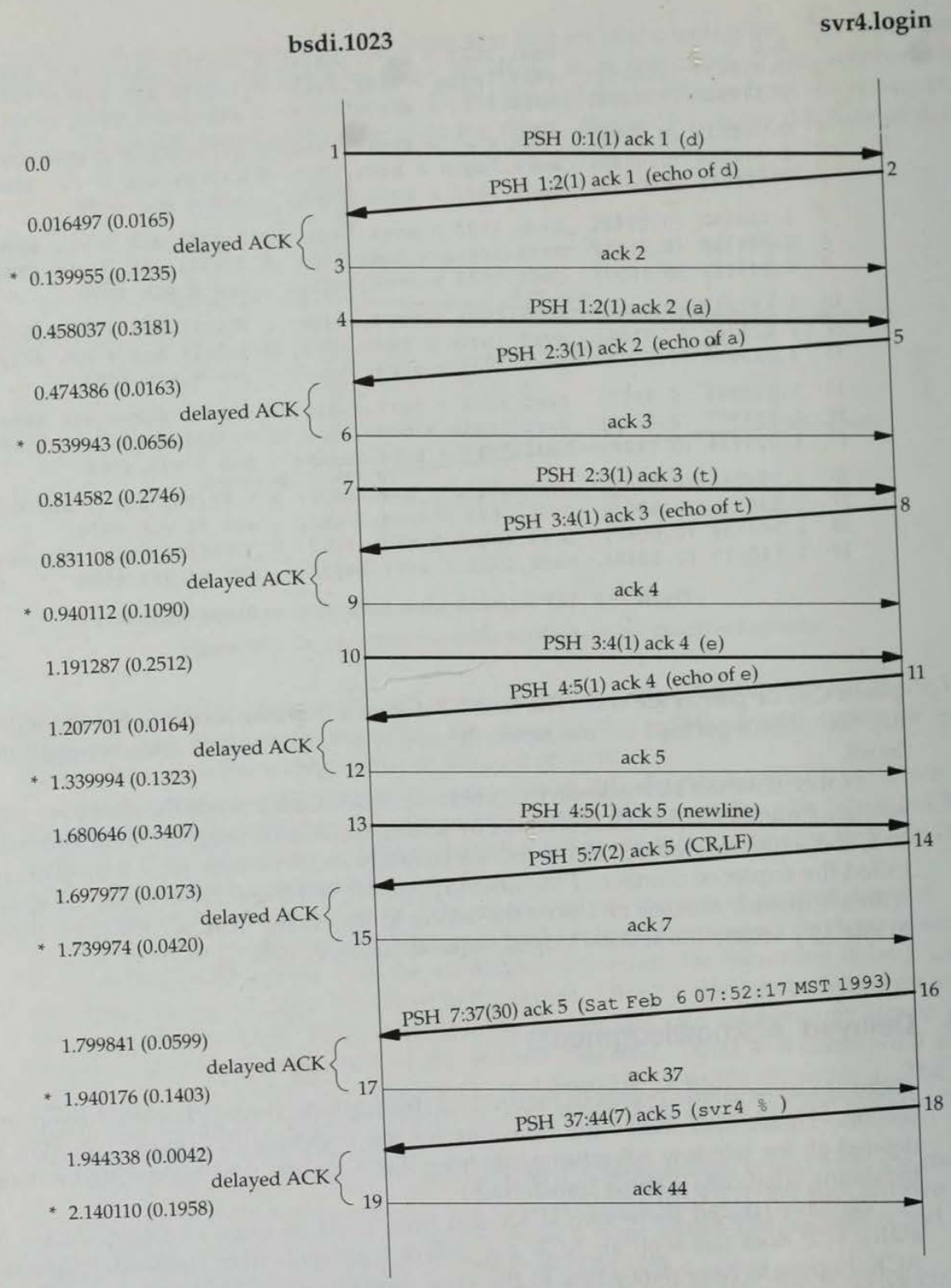


Figure 19.3 Time line of data flow for `date` command typed on an `rlogin` connection.

If we look at the time differences between `bsdi` receiving the data and sending the ACK, they appear to be random: 123.5, 65.6, 109.0, 132.3, 42.0, 140.3, and 195.8 ms. Look instead at the actual times (starting from 0) when the ACKs are sent: 139.9, 539.9, 940.1, 1339.9, 1739.9, 1940.1, and 2140.1 ms. (We have marked these with an asterisk to the left of the time in Figure 19.3.) There is a multiple of 200 ms between these times. What is happening here is that TCP has a timer that goes off every 200 ms, but it goes off at fixed points in time—every 200 ms relative to when the kernel was bootstrapped. Since the data being acknowledged arrives randomly (at times 16.4, 474.3, 831.1, etc.), TCP asks to be notified the next time the kernel's 200-ms timer expires. This can be anywhere from 1 to 200 ms in the future.

If we look at how long it takes `svr4` to generate the echo of each character it receives, the times are 16.5, 16.3, 16.5, 16.4, and 17.3 ms. Since this time is less than 200 ms, we never see a delayed ACK on that side. There is always data ready to be sent before the delayed ACK timer expires. (We could still see a delayed ACK if the wait period, about 16 ms, crosses one of the kernel's 200-ms clock tick boundaries. We just don't see any of these in this example.)

We saw this same scenario in Figure 18.7 with the 500-ms TCP timer used when detecting a timeout. Both TCP timers, the 200- and 500-ms timers, go off at times relative to when the kernel was bootstrapped. Whenever TCP sets a timer, it can go off anywhere between 1–200 or 1–500 ms in the future.

> The Host Requirements RFC states that TCP should implement a delayed ACK but the delay must be less than 500 ms.

19.4 Nagle Algorithm

We saw in the previous section that 1 byte at a time normally flows from the client to the server across an Rlogin connection. This generates 41-byte packets: 20 bytes for the IP header, 20 bytes for the TCP header, and 1 byte of data. These small packets (called *tinygrams*) are normally not a problem on LANs, since most LANs are not congested, but these tinygrams can add to congestion on wide area networks. A simple and elegant solution was proposed in RFC 896 [Nagle 1984], called the *Nagle algorithm*.

This algorithm says that a TCP connection can have only one outstanding small segment that has not yet been acknowledged. No additional small segments can be sent until the acknowledgment is received. Instead, small amounts of data are collected by TCP and sent in a single segment when the acknowledgment arrives. The beauty of this algorithm is that it is self-clocking: the faster the ACKs come back, the faster the data is sent. But on a slow WAN, where it is desired to reduce the number of tinygrams, fewer segments are sent. (We'll see in Section 22.3 that the definition of "small" is less than the segment size.)

We saw in Figure 19.3 that the round-trip time on an Ethernet for a single byte to be sent, acknowledged, and echoed averaged around 16 ms. To generate data faster than this we would have to be typing more than 60 characters per second. This means we rarely encounter this algorithm when sending data between two hosts on a LAN.

Things change, however, when the round-trip time (RTT) increases, typically across a WAN. Let's look at an Rlogin connection between our host `slip` and the host `vangogh.cs.berkeley.edu`. To get out of our network (see inside front cover), two SLIP links must be traversed, and then the Internet is used. We expect much longer round-trip times. Figure 19.4 shows the time line of some data flow while characters were being typed quickly on the client (similar to a fast typist). (We have removed the type-of-service information, but have left in the window size advertisements.)

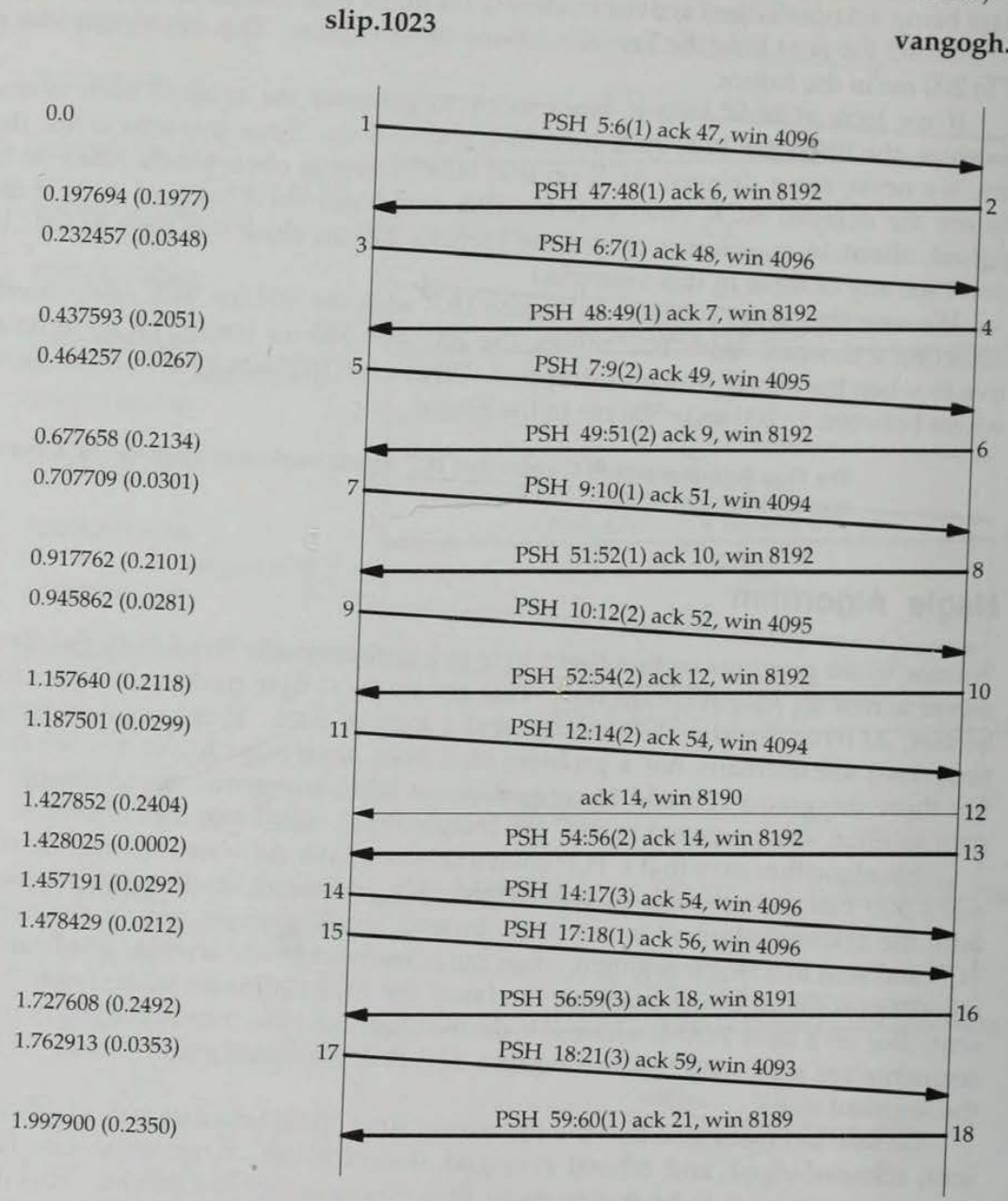


Figure 19.4 Data flow using `rlogin` between `slip` and `vangogh.cs.berkeley.edu`.

The first thing we notice, comparing Figure 19.4 with Figure 19.3, is the lack of delayed ACKs from `slip` to `vangogh`. This is because there is always data ready to send before the delayed ACK timer expires.

Next, notice the various amounts of data being sent from the left to the right: 1, 1, 2, 1, 2, 2, 3, 1, and 3 bytes. This is because the client is collecting the data to send, but doesn't send it until the previously sent data has been acknowledged. By using the Nagle algorithm only nine segments were used to send 16 bytes, instead of 16 segments.

Segments 14 and 15 appear to contradict the Nagle algorithm, but we need to look at the sequence numbers to see what's really happening. Segment 14 is in response to the ACK received in segment 12, since the acknowledged sequence number is 54. But before this data segment is sent by the client, segment 13 arrives from the server. Segment 15 contains the ACK of segment 13, sequence number 56. So the client is obeying the Nagle algorithm, even though we see two back-to-back data segments from the client to the server.

Also notice in Figure 19.4 that one delayed ACK is present, but it's from the server to the client (segment 12). We are assuming this is a delayed ACK since it contains no data. The server must have been busy at this time, so that the Rlogin server was not able to echo the character before the server's delayed ACK timer expired.

Finally, look at the amounts of data and the sequence numbers in the final two segments. The client sends 3 bytes of data (numbered 18, 19, and 20), then the server acknowledges these 3 bytes (the ACK of 21 in the final segment) but sends back only 1 byte (numbered 59). What's happening here is that the server's TCP is acknowledging the 3 bytes of data once it has received them correctly, but it won't have the echo of these 3 bytes ready to send back until the Rlogin server sends them. This shows that TCP can acknowledge received data before the application has read and processed that data. The TCP acknowledgment just means TCP has correctly received the data. We also have an indication that the server process has not read these 3 bytes of data because the advertised window in the final segment is 8189, not 8192.

Disabling the Nagle Algorithm

There are times when the Nagle algorithm needs to be turned off. The classic example is the X Window System server (Section 30.5): small messages (mouse movements) must be delivered without delay to provide real-time feedback for interactive users doing certain operations.

Here we'll show another example that's easier to demonstrate—typing one of the terminal's special function keys during an interactive login. The function keys normally generate multiple bytes of data, often beginning with the ASCII escape character. If TCP gets the data 1 byte at a time, it's possible for it to send the first byte (the ASCII ESC) and then hold the remaining bytes of the sequence waiting for the ACK of this byte. But when the server receives this first byte it doesn't generate an echo until the remaining bytes are received. This often triggers the delayed ACK algorithm on the server, meaning that the remaining bytes aren't sent for up to 200 ms. This can lead to noticeable delays to the interactive user.

The sockets API uses the `TCP_NODELAY` socket option to disable the Nagle algorithm.

The Host Requirements RFC states that TCP should implement the Nagle algorithm but there must be a way for an application to disable it on an individual connection.

An Example

We can see this interaction between the Nagle algorithm and keystrokes that generate multiple bytes. We establish an Rlogin connection from our host `slip` to the host `vangogh.cs.berkeley.edu`. We then type the F1 function key, which generates 3 bytes: an escape, a left bracket, and an M. We then type the F2 function key, which generates another 3 bytes. Figure 19.5 shows the `tcpdump` output. (We have removed the type-of-service information and the window advertisements.)

```
                                    type F1 key
 1  0.0                             slip.1023 > vangogh.login: P 1:2(1) ack 2
 2  0.250520 (0.2505)               vangogh.login > slip.1023: P 2:4(2) ack 2
 3  0.251709 (0.0012)               slip.1023 > vangogh.login: P 2:4(2) ack 4
 4  0.490344 (0.2386)               vangogh.login > slip.1023: P 4:6(2) ack 4
 5  0.588694 (0.0984)               slip.1023 > vangogh.login: . ack 6
                                    type F2 key
 6  2.836830 (2.2481)               slip.1023 > vangogh.login: P 4:5(1) ack 6
 7  3.132388 (0.2956)               vangogh.login > slip.1023: P 6:8(2) ack 5
 8  3.133573 (0.0012)               slip.1023 > vangogh.login: P 5:7(2) ack 8
 9  3.370346 (0.2368)               vangogh.login > slip.1023: P 8:10(2) ack 7
10  3.388692 (0.0183)               slip.1023 > vangogh.login: . ack 10
```

Figure 19.5 Watching the Nagle algorithm when typing characters that generate multiple bytes of data.

Figure 19.6 shows the time line for this exchange. At the bottom of this figure we show the 6 bytes going from the client to the server with their sequence numbers, and the 8 bytes of echo being returned.

When the first byte of input is read by the `rlogin` client and written to TCP, it is sent by itself as segment 1. This is the first of the 3 bytes generated by the F1 key. Its echo is returned in segment 2, and only then are the next 2 bytes sent (segment 3). The echo of the second 2 bytes is received in segment 4 and acknowledged in segment 5.

The reason the echo of the first byte occupies 2 bytes (segment 2) is because the ASCII escape character is echoed as 2 bytes: a caret and a left bracket. The next 2 bytes of input, a left bracket and an M, are echoed as themselves.

The same exchange occurs when the next special function key is typed (segments 6–10). As we expect, the time difference between segments 5 and 10 (`slip` sending the acknowledgment of the echo) is a multiple of 200 ms, since both ACKs are delayed.

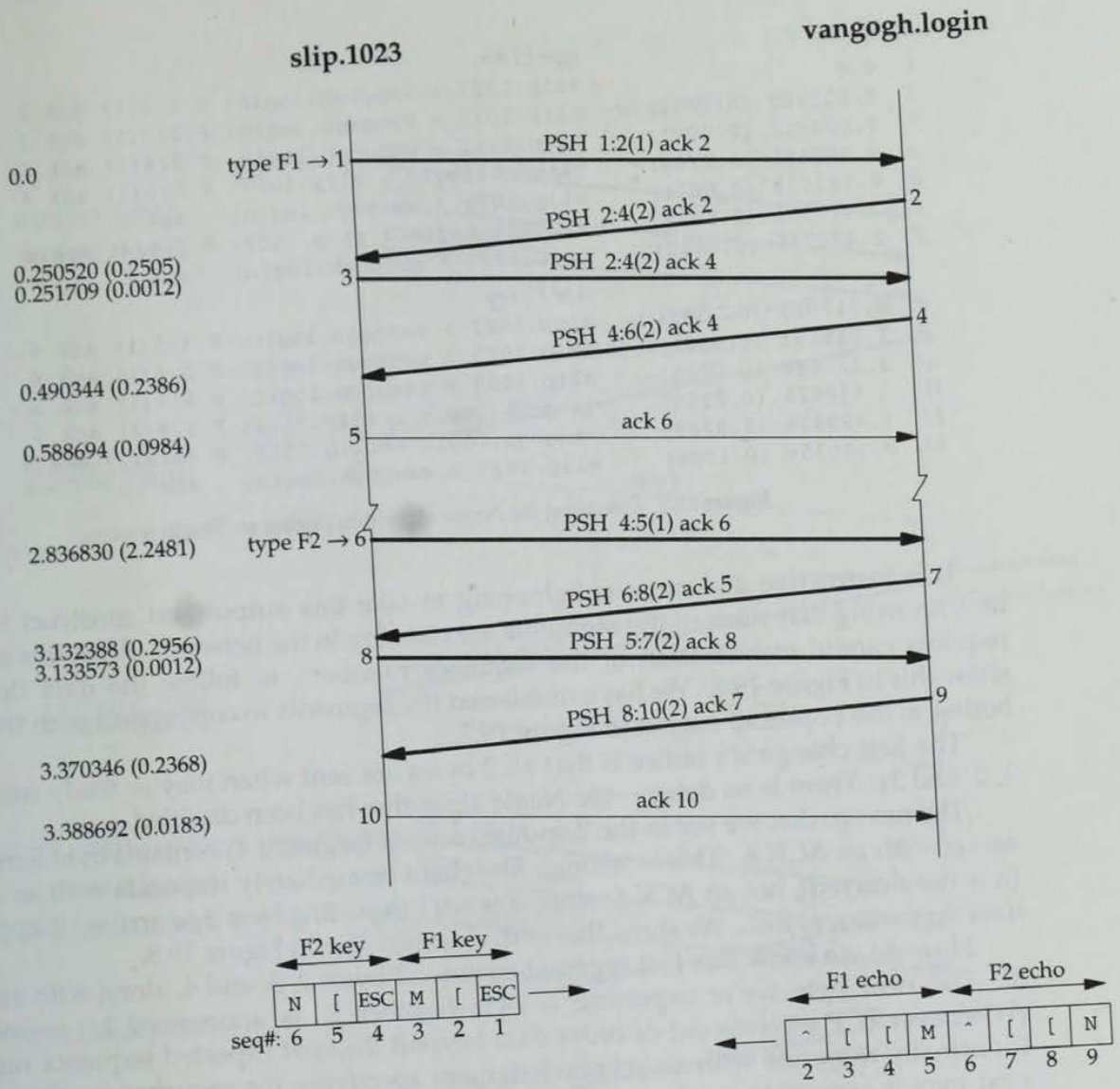


Figure 19.6 Time line for Figure 19.5 (watching the Nagle algorithm).

We now repeat this same example using a version of `rlogin` that has been modified to turn off the Nagle algorithm. Figure 19.7 shows the `tcpdump` output. (Again, we have deleted the type-of-service information and the window advertisements.)

```
                                    type F1 key
 1  0.0                 slip.1023 > vangogh.login: P 1:2(1) ack 2
 2  0.002163 (0.0022)   slip.1023 > vangogh.login: P 2:3(1) ack 2
 3  0.004218 (0.0021)   slip.1023 > vangogh.login: P 3:4(1) ack 2
 4  0.280621 (0.2764)   vangogh.login > slip.1023: P 5:6(1) ack 4
 5  0.281738 (0.0011)   slip.1023 > vangogh.login: . ack 2
 6  2.477561 (2.1958)   vangogh.login > slip.1023: P 2:6(4) ack 4
 7  2.478735 (0.0012)   slip.1023 > vangogh.login: . ack 6
                                    type F2 key
 8  3.217023 (0.7383)   slip.1023 > vangogh.login: P 4:5(1) ack 6
 9  3.219165 (0.0021)   slip.1023 > vangogh.login: P 5:6(1) ack 6
10  3.221688 (0.0025)   slip.1023 > vangogh.login: P 6:7(1) ack 6
11  3.460626 (0.2389)   vangogh.login > slip.1023: P 6:8(2) ack 5
12  3.489414 (0.0288)   vangogh.login > slip.1023: P 8:10(2) ack 7
13  3.640356 (0.1509)   slip.1023 > vangogh.login: . ack 10
```

Figure 19.7 Disabling the Nagle algorithm during an Rlogin session.

It is instructive and more enlightening to take this output and construct the time line, knowing that some of the segments are crossing in the network. Also, this example requires careful examination of the sequence numbers, to follow the data flow. We show this in Figure 19.8. We have numbered the segments to correspond with the numbering in the `tcpdump` output in Figure 19.7.

The first change we notice is that all 3 bytes are sent when they're ready (segments 1, 2, and 3). There is no delay—the Nagle algorithm has been disabled.

The next packet we see in the `tcpdump` output (segment 4) contains byte 5 from the server with an ACK 4. This is wrong. The client immediately responds with an ACK 2 (it is not delayed), not an ACK 6, since it wasn't expecting byte 5 to arrive. It appears a data segment was lost. We show this with a dashed line in Figure 19.8.

How do we know this lost segment contained bytes 2, 3, and 4, along with an ACK 3? The next byte we're expecting is byte number 2, as announced by segment 5. (Whenever TCP receives out-of-order data beyond the next expected sequence number, it normally responds with an acknowledgment specifying the sequence number of the next byte it expects to receive.) Also, since the missing segment contained bytes 2, 3, and 4, it means the server must have received segment 2, so the missing segment must have specified an ACK 3 (the sequence number of the next byte the server is expecting to receive.) Finally, notice that the retransmission, segment 6, contains data from the missing segment and segment 4. This is called *repacketization*, and we'll discuss it more in Section 21.11.

Returning to our discussion of disabling the Nagle algorithm, we can see the 3 bytes of the next special function key that we type is sent as three individual segments (8, 9, and 10). This time the server echoes the byte in segment 8 first (segment 11), and then echoes the bytes in segments 9 and 10 (segment 12).

What we've seen in this example is that the default use of the Nagle algorithm can cause additional delays when multibyte keystrokes are entered while running an interactive application across a WAN.

We return to the topic of timeout and retransmission in Chapter 21.

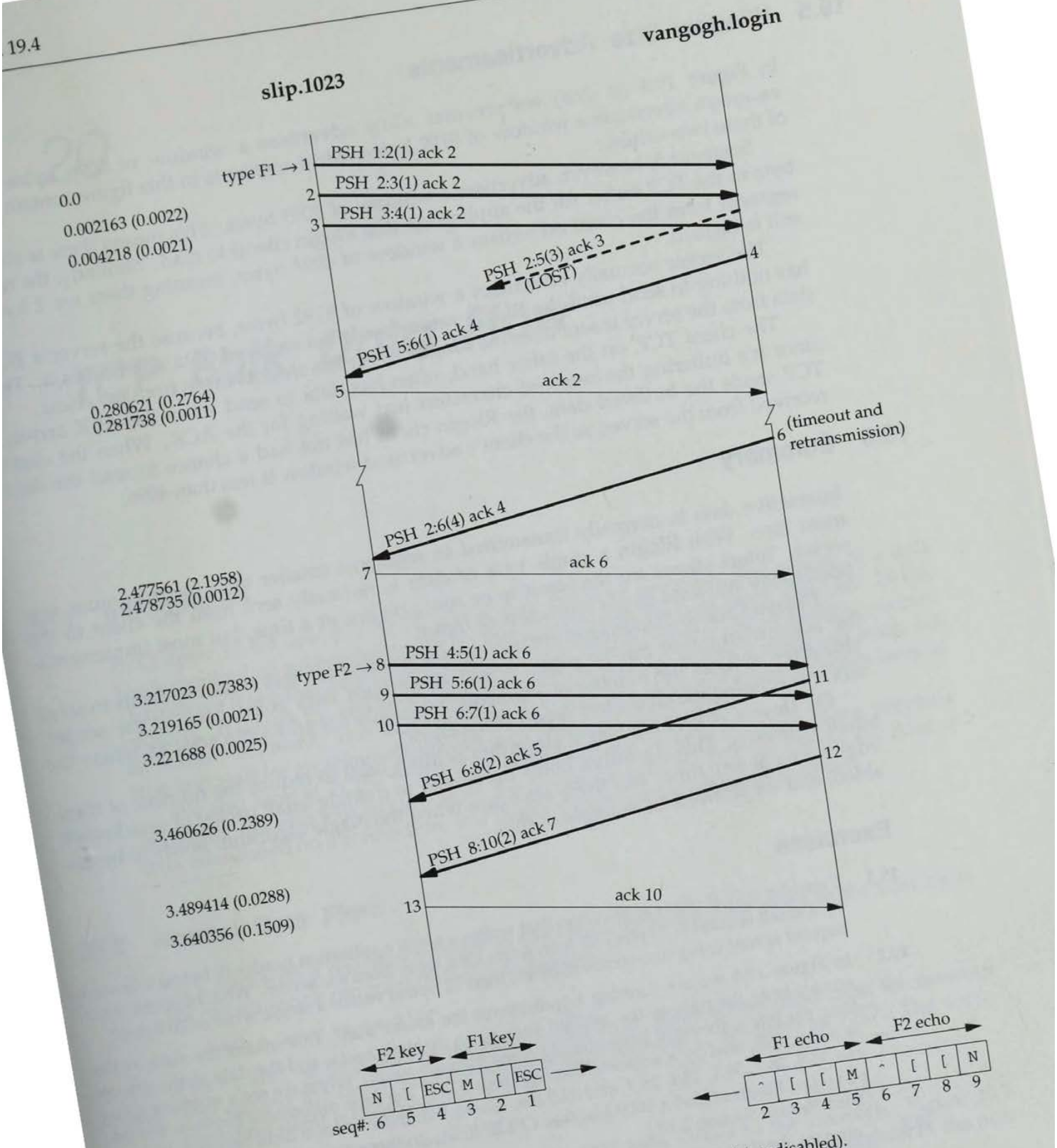


Figure 19.8 Time line for Figure 19.7 (Nagle algorithm disabled).

19.5 Window Size Advertisements

In Figure 19.4 (p. 268) we see that `slip` advertises a window of 4096 bytes and `vangogh` advertises a window of 8192 bytes. Most segments in this figure contain one of these two values.

Segment 5, however, advertises a window of 4095 bytes. This means there is still 1 byte in the TCP buffer for the application (the Rlogin client) to read. Similarly, the next segment from the client advertises a window of 4094 bytes, meaning there are 2 bytes still to be read.

The server normally advertises a window of 8192 bytes, because the server's TCP has nothing to send until the Rlogin server reads the received data and echoes it. The data from the server is sent after the Rlogin server has read its input from the client.

The client TCP, on the other hand, often has data to send when the ACK arrives, since it's buffering the received characters just waiting for the ACK. When the client TCP sends the buffered data, the Rlogin client has not had a chance to read the data received from the server, so the client's advertised window is less than 4096.

19.6 Summary

Interactive data is normally transmitted in segments smaller than the maximum segment size. With Rlogin a single byte of data is normally sent from the client to the server. Telnet allows for the input to be sent one line at a time, but most implementations today still send single characters of input.

Delayed acknowledgments are used by the receiver of these small segments to see if the acknowledgment can be piggybacked along with data going back to the sender. This often reduces the number of segments, especially for an Rlogin session, where the server is echoing the characters typed at the client.

On slower WANs the Nagle algorithm is often used to reduce the number of these small segments. This algorithm limits the sender to a single small packet of unacknowledged data at any time. But there are times when the Nagle algorithm needs to be disabled, and we showed an example of this.

Exercises

19.1 Consider a TCP client application that writes a small application header (8 bytes) followed by a small request (12 bytes). It then waits for a reply from the server. What happens if the request is sent using two writes (8 bytes, then 12 bytes) versus a single write of 20 bytes?

19.2 In Figure 19.4 we are running `tcpdump` on the router `sun`. This means the data in the arrows from the right to the left still have to go through `bsdi`, and the data in the arrows from the left to the right have already come through `bsdi`. When we see a segment going to `slip`, followed by a segment coming from `slip`, the time differences between the two are: 34.8, 26.7, 30.1, 28.1, 29.9, and 35.3 ms. Given that there are two links between `sun` and `slip` (an Ethernet and a 9600 bits/sec CSLIP link), do these time differences make sense? (*Hint*: Reread Section 2.10.)

19.3 Compare the time required to send a special function key and have it acknowledged using the Nagle algorithm (Figure 19.6) and with the algorithm disabled (Figure 19.8).

20

TCP Bulk Data Flow

20.1 Introduction

In Chapter 15 we saw that TFTP uses a stop-and-wait protocol. The sender of a data block required an acknowledgment for that block before the next block was sent. In this chapter we'll see that TCP uses a different form of flow control called a *sliding window* protocol. It allows the sender to transmit multiple packets before it stops and waits for an acknowledgment. This leads to faster data transfer, since the sender doesn't have to stop and wait for an acknowledgment each time a packet is sent.

We also look at TCP's PUSH flag, something we've seen in many of the previous examples. We also look at slow start, the technique used by TCP for getting the flow of data established on a connection, and then we examine bulk data throughput.

20.2 Normal Data Flow

Let's start with a one-way transfer of 8192 bytes from the host `svr4` to the host `bsdi`. We run our `sock` program on `bsdi` as the server:

```
bsdi % sock -i -s 7777
```

The `-i` and `-s` flags tell the program to run as a "sink" server (read from the network and discard the data), and the server's port number is specified as 7777. The corresponding client is then run as:

```
svr4 % sock -i -n8 bsdi 7777
```

This causes the client to perform eight 1024-byte writes to the network. Figure 20.1 shows the time line for this exchange. We have left the first three segments in the output to show the MSS values for each end.

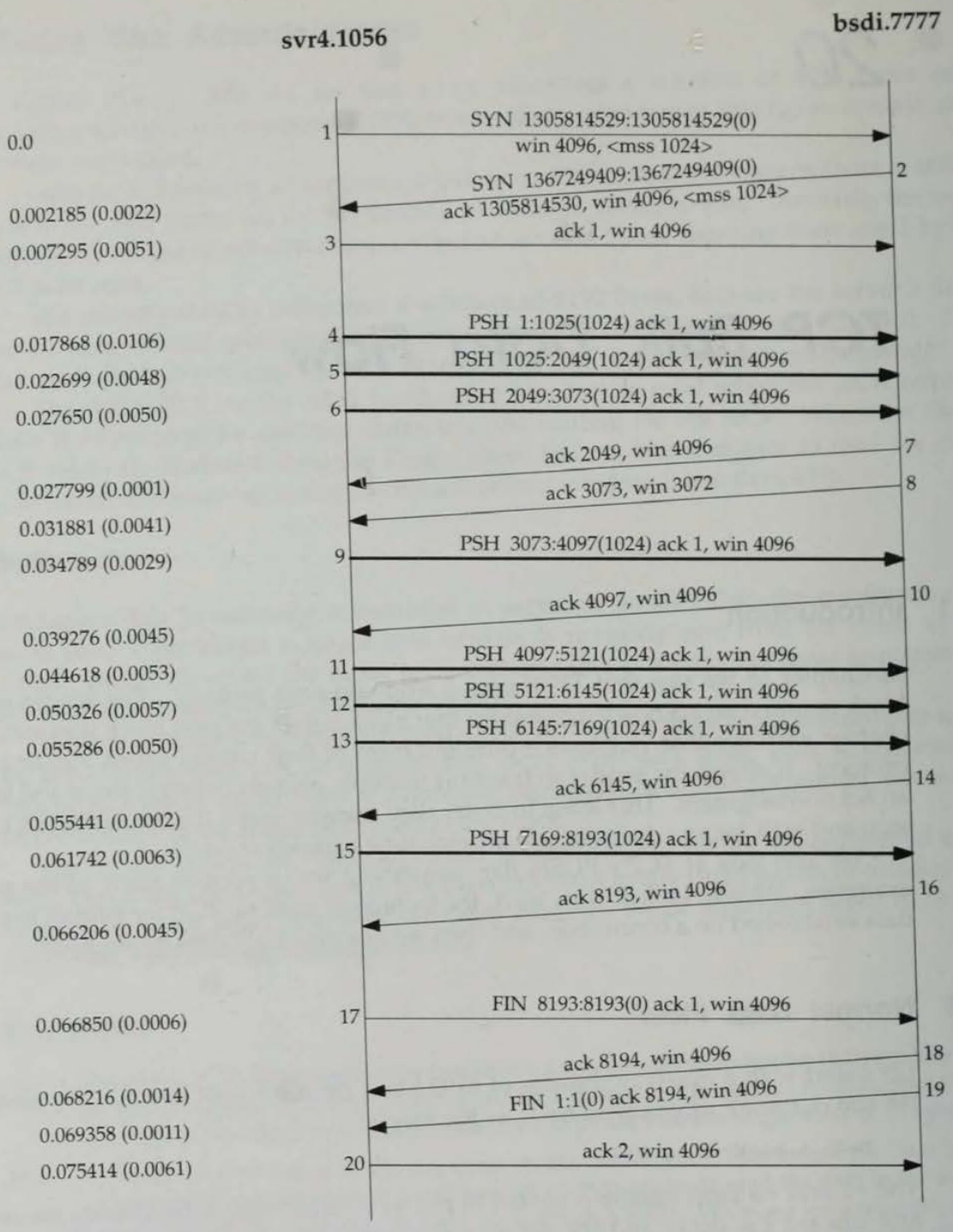


Figure 20.1 Transfer of 8192 bytes from `svr4` to `bsdi`.

The sender transmits three data segments (4–6) first. The next segment (7) acknowledges the first two data segments only. We know this because the acknowledged sequence number is 2049, not 3073.

Segment 7 specifies an ACK of 2049 and not 3073 for the following reason. When a packet arrives it is initially processed by the device driver's interrupt service routine and then placed onto IP's input queue. The three segments 4, 5, and 6 arrive one after the other and are placed onto IP's input queue in the received order. IP will pass them to TCP in the same order. When TCP processes segment 4, the connection is marked to generate a delayed ACK. TCP processes the next segment (5) and since TCP now has two outstanding segments to ACK, the ACK of 2049 is generated (segment 7), and the delayed ACK flag for this connection is turned off. TCP processes the next input segment (6) and the connection is again marked for a delayed ACK. Before segment 9 arrives, however, it appears the delayed ACK timer goes off, and the ACK of 3073 (segment 8) is generated. Segment 8 advertises a window of 3072 bytes, implying that there are still 1024 bytes of data in the TCP receive buffer that the application has not read.

Segments 11–16 show the "ACK every other segment" strategy that is common. Segments 11, 12, and 13 arrive and are placed on IP's input queue. When segment 11 is processed by TCP the connection is marked for a delayed ACK. When segment 12 is processed, an ACK is generated (segment 14) for segments 11 and 12, and the delayed ACK flag for this connection is turned off. Segment 13 causes the connection to be marked again for a delayed ACK but before the timer goes off, segment 15 is processed, causing the ACK (segment 16) to be sent immediately.

It is important to notice that the ACK in segments 7, 14, and 16 acknowledge two received segments. With TCP's sliding-window protocol the receiver does not have to acknowledge every received packet. With TCP, the ACKs are cumulative—they acknowledge that the receiver has correctly received all bytes up through the acknowledged sequence number minus one. In this example three of the ACKs acknowledge 2048 bytes of data and two acknowledge 1024 bytes of data. (This ignores the ACKs in the connection establishment and termination.)

What we are watching with `tcpdump` are the dynamics of TCP in action. The ordering of the packets that we see on the wire depends on many factors, most of which we have no control over: the sending TCP implementation, the receiving TCP implementation, the reading of data by the receiving process (which depends on the process scheduling by the operating system), and the dynamics of the network (i.e., Ethernet collisions and backoffs). There is no single correct way for two TCPs to exchange a given amount of data.

To show how things can change, Figure 20.2 shows another time line for the same exchange of data between the same two hosts, captured a few minutes after the one in Figure 20.1.

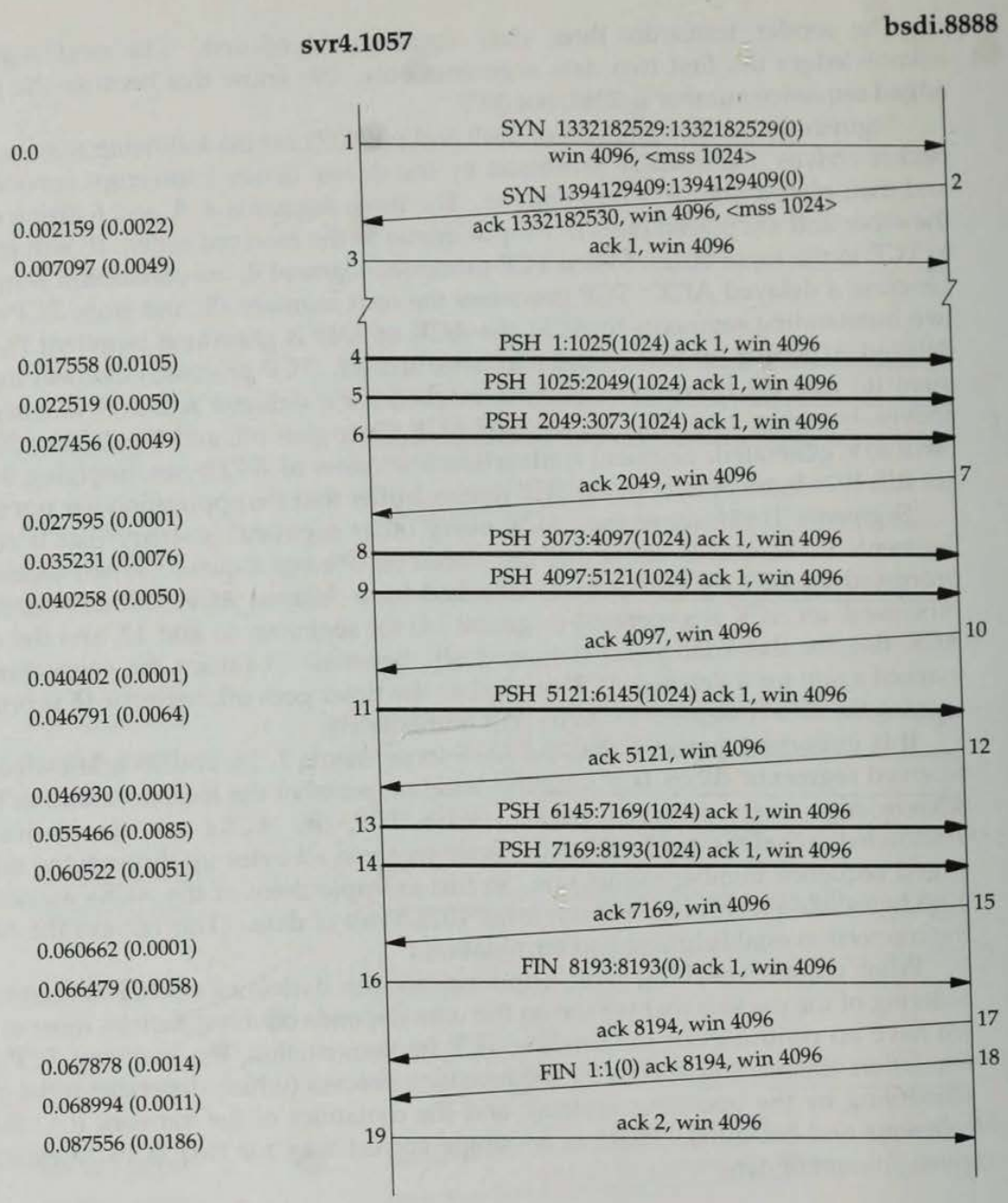


Figure 20.2 Another transfer of 8192 bytes from `svr4` to `bsdi`.

A few things have changed. This time the receiver does not send an ACK of 3073; instead it waits and sends the ACK of 4097. The receiver sends only four ACKs (segments 7, 10, 12, and 15): three of these are for 2048 bytes and one for 1024 bytes. The ACK of the final 1024 bytes of data appears in segment 17, along with the ACK of the FIN. (Compare segment 17 in this figure with segments 16 and 18 in Figure 20.1.)

Fast Sender, Slow Receiver

Figure 20.3 shows another time line, this time from a fast sender (a Sparc) to a slow receiver (an 80386 with a slow Ethernet card). The dynamics are different again.

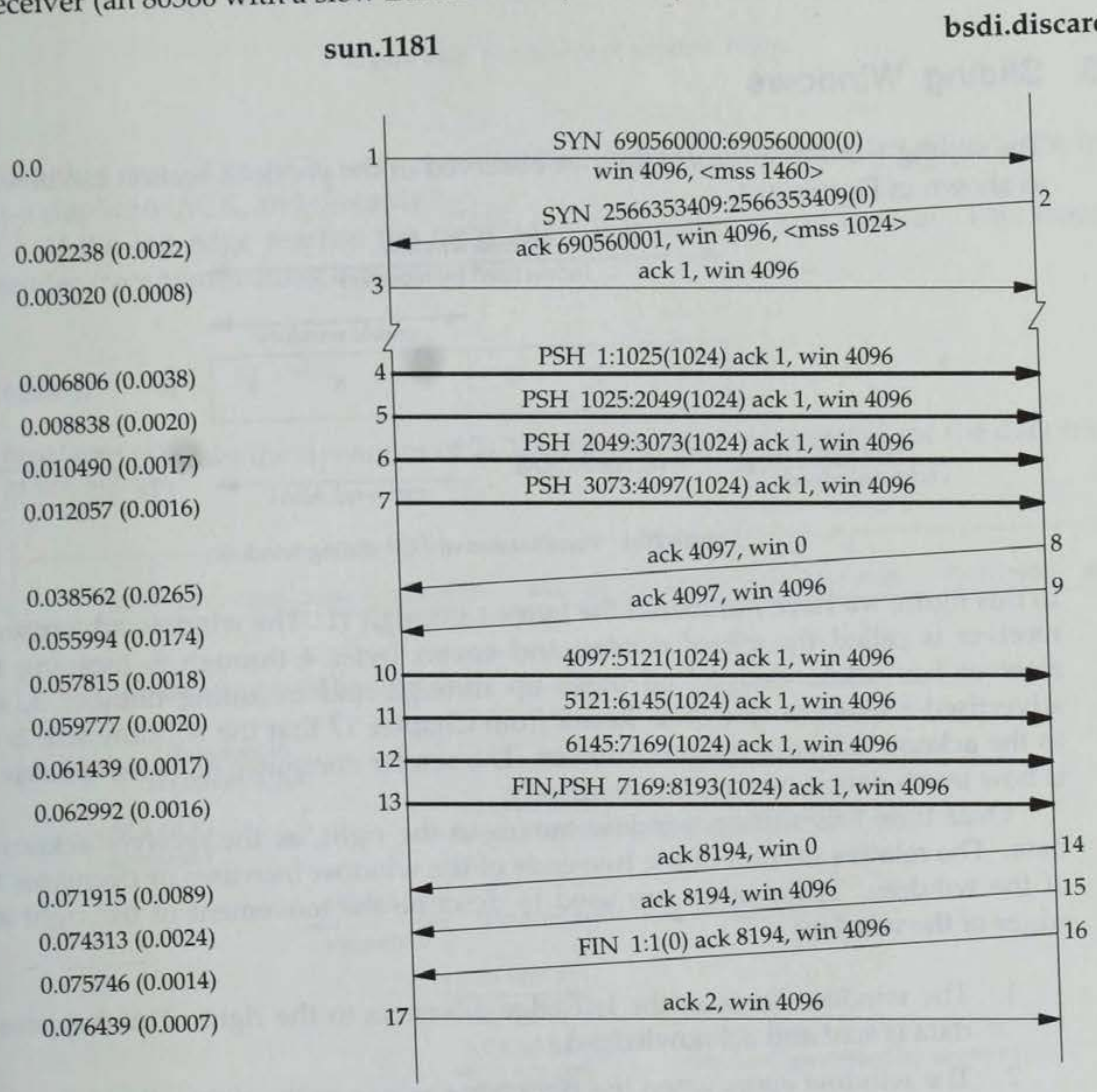


Figure 20.3 Sending 8192 bytes from a fast sender to a slow receiver.

The sender transmits four back-to-back data segments (4–7) to fill the receiver's window. The sender then stops and waits for an ACK. The receiver sends the ACK (segment 8) but the advertised window is 0. This means the receiver has all the data, but it's all in the receiver's TCP buffers, because the application hasn't had a chance to read the data. Another ACK (called a *window update*) is sent 17.4 ms later, announcing that the receiver can now receive another 4096 bytes. Although this looks like an ACK, it is called a window update because it does not acknowledge any new data, it just advances the right edge of the window.

The sender transmits its final four segments (10–13), again filling the receiver's window. Notice that segment 13 contains two flag bits: PUSH and FIN. This is followed by another two ACKs from the receiver. Both of these acknowledge the final 4096 bytes of data (bytes 4097 through 8192) and the FIN (numbered 8193).

20.3 Sliding Windows

The sliding window protocol that we observed in the previous section can be visualized as shown in Figure 20.4.

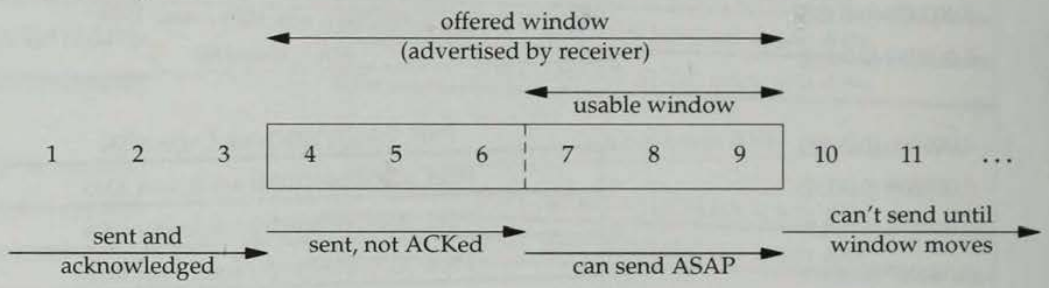


Figure 20.4 Visualization of TCP sliding window.

In this figure we have numbered the bytes 1 through 11. The window advertised by the receiver is called the *offered window* and covers bytes 4 through 9, meaning that the receiver has acknowledged all bytes up through and including number 3, and has advertised a window size of 6. Recall from Chapter 17 that the window size is relative to the acknowledged sequence number. The sender computes its *usable window*, which is how much data it can send immediately.

Over time this sliding window moves to the right, as the receiver acknowledges data. The relative motion of the two ends of the window increases or decreases the size of the window. Three terms are used to describe the movement of the right and left edges of the window.

1. The window *closes* as the left edge advances to the right. This happens when data is sent and acknowledged.
2. The window *opens* when the right edge moves to the right, allowing more data to be sent. This happens when the receiving process on the other end reads acknowledged data, freeing up space in its TCP receive buffer.
3. The window *shrinks* when the right edge moves to the left. The Host Requirements RFC strongly discourages this, but TCP must be able to cope with a peer that does this. Section 22.3 shows an example when one side would like to shrink the window by moving the right edge to the left, but cannot.

Figure 20.5 shows these three terms. The left edge of the window cannot move to the left, because this edge is controlled by the acknowledgment number received from

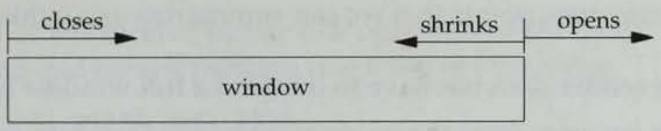


Figure 20.5 Movement of window edges.

the other end. If an ACK were received that implied moving the left edge to the left, it is a duplicate ACK, and discarded.

If the left edge reaches the right edge, it is called a *zero window*. This stops the sender from transmitting any data.

An Example

Figure 20.6 shows the dynamics of TCP's sliding window protocol for the data transfer in Figure 20.1.

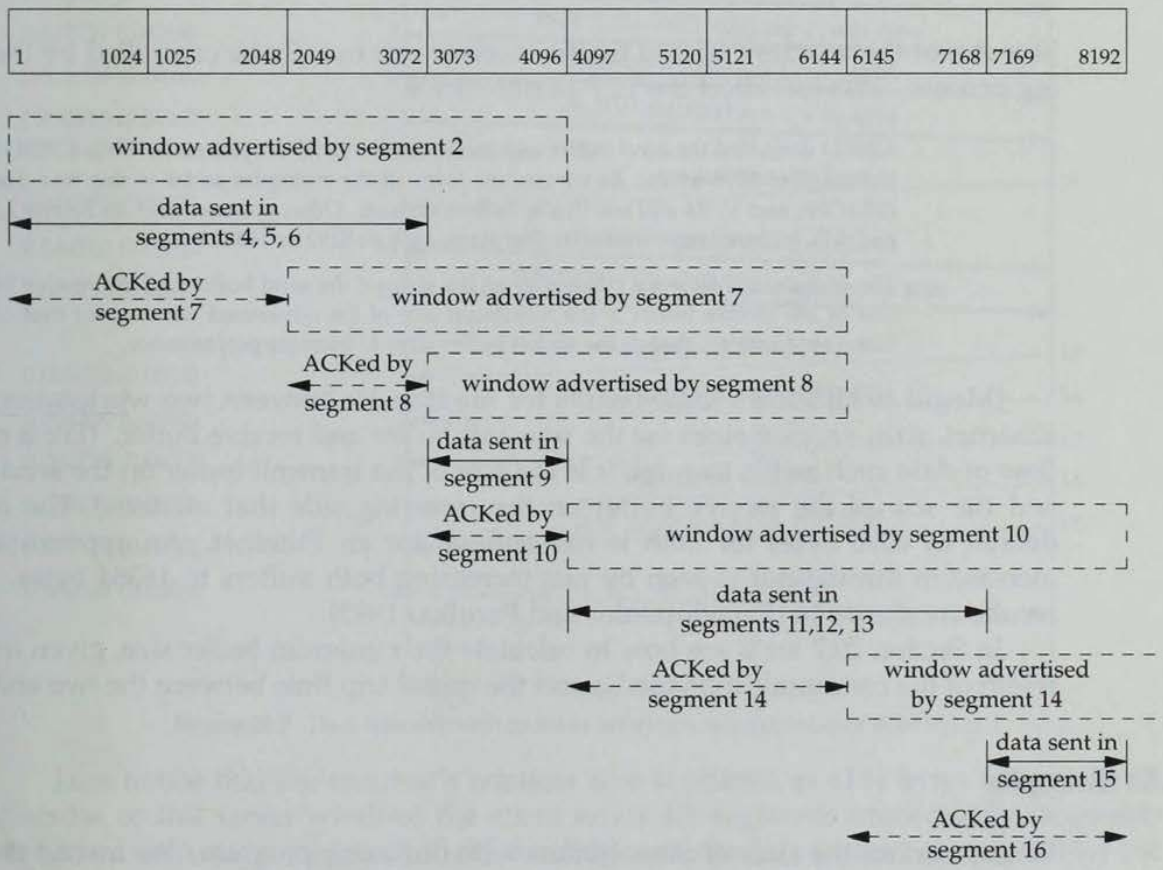


Figure 20.6 Sliding window protocol for Figure 20.1.

There are numerous points that we can summarize using this figure as an example.

1. The sender does not have to transmit a full window's worth of data.
2. One segment from the receiver acknowledges data and slides the window to the right. This is because the window size is relative to the acknowledged sequence number.
3. The size of the window can decrease, as shown by the change from segment 7 to segment 8, but the right edge of the window must not move leftward.
4. The receiver does not have to wait for the window to fill before sending an ACK. We saw earlier that many implementations send an ACK for every two segments that are received.

We'll see more examples of the dynamics of the sliding window protocol in later examples.

20.4 Window Size

The size of the window offered by the receiver can usually be controlled by the receiving process. This can affect the TCP performance.

> 4.2BSD defaulted the send buffer and receive buffer to 2048 bytes each. With 4.3BSD both were increased to 4096 bytes. As we can see from all the examples so far in this text, SunOS 4.1.3, BSD/386, and SVR4 still use this 4096-byte default. Other systems, such as Solaris 2.2, 4.4BSD, and AIX 3.2, use larger default buffer sizes, such as 8192 or 16384 bytes.
>
> The sockets API allows a process to set the sizes of the send buffer and the receive buffer. The size of the receive buffer is the maximum size of the advertised window for that connection. Some applications change the socket buffer sizes to increase performance.

[Mogul 1993] shows some results for file transfer between two workstations on an Ethernet, with varying sizes for the transmit buffer and receive buffer. (For a one-way flow of data such as file transfer, it is the size of the transmit buffer on the sending side and the size of the receive buffer on the receiving side that matters.) The common default of 4096 bytes for both is not optimal for an Ethernet. An approximate 40% increase in throughput is seen by just increasing both buffers to 16384 bytes. Similar results are shown in [Papadopoulos and Parulkar 1993].

In Section 20.7 we'll see how to calculate the minimum buffer size, given the bandwidth of the communication media and the round-trip time between the two ends.

An Example

We can control the sizes of these buffers with our `sock` program. We invoke the server as:

```
bsdi % sock -i -s -R6144 5555
```

which sets the size of the receive buffer (-R option) to 6144 bytes. We then start the client on the host sun and have it perform one write of 8192 bytes:

```
sun % sock -i -n1 -w8192 bsdi 5555
```

Figure 20.7 shows the results.

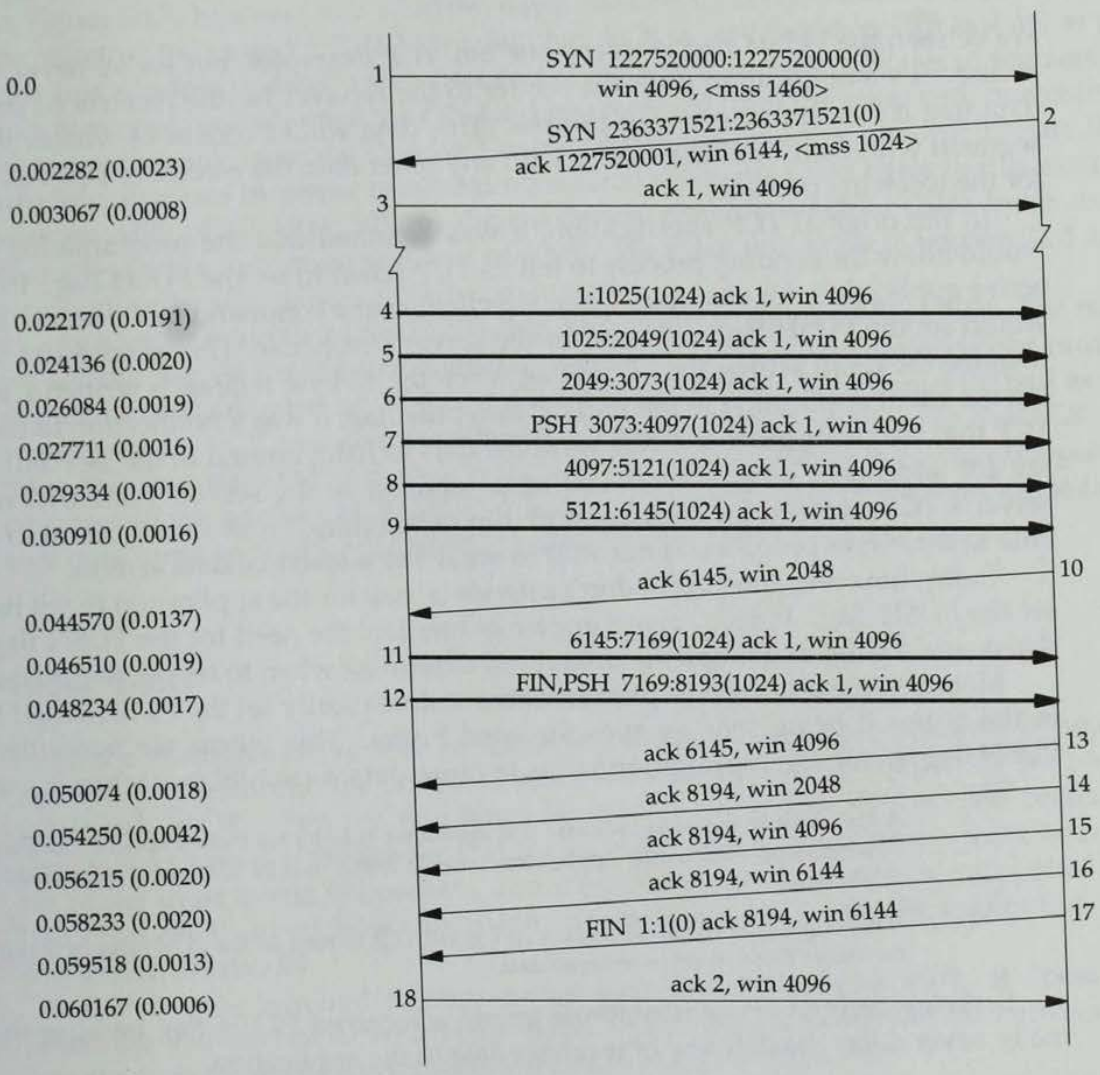


Figure 20.7 Data transfer with receiver offering a window size of 6144 bytes.

First notice that the receiver's window size is offered as 6144 bytes in segment 2. Because of this larger window, the client sends six segments immediately (segments 4–9), and then stops. Segment 10 acknowledges all the data (bytes 1 through 6144) but offers a window of only 2048, probably because the receiving application hasn't had a chance to read more than 2048 bytes. Segments 11 and 12 complete the data transfer from the client, and this final data segment also carries the FIN flag.

Segment 13 contains the same acknowledgment sequence number as segment 10, but advertises a larger window. Segment 14 acknowledges the final 2048 bytes of data and the FIN, and segments 15 and 16 just advertise a larger window. Segments 17 and 18 complete the normal close.

20.5 PUSH Flag

We've seen the PUSH flag in every one our TCP examples, but we've never described its use. It's a notification from the sender to the receiver for the receiver to pass all the data that it has to the receiving process. This data would consist of whatever is in the segment with the PUSH flag, along with any other data the receiving TCP has collected for the receiving process.

In the original TCP specification, it was assumed that the programming interface would allow the sending process to tell its TCP when to set the PUSH flag. In an interactive application, for example, when the client sent a command to the server, the client would set the PUSH flag and wait for the server's response. (In Exercise 19.1 we could imagine the client setting the PUSH flag when the 12-byte request is written.) By allowing the client application to tell its TCP to set the flag, it was a notification to the client's TCP that the client process didn't want the data to hang around in the TCP buffer, waiting for additional data, before sending a segment to the server. Similarly, when the server's TCP received the segment with the PUSH flag, it was a notification to pass the data to the server process and not wait to see if any additional data arrives.

Today, however, most APIs don't provide a way for the application to tell its TCP to set the PUSH flag. Indeed, many implementors feel the need for the PUSH flag is outdated, and a good TCP implementation can determine when to set the flag by itself.

Most Berkeley-derived implementations automatically set the PUSH flag if the data in the segment being sent empties the send buffer. This means we normally see the PUSH flag set for each application write, because data is usually sent when it's written.

> A comment in the code indicates this algorithm is to please those implementations that only pass received data to the application when a buffer fills or a segment is received with the PUSH flag.
>
> It is not possible using the sockets API to tell TCP to turn on the PUSH flag or to tell whether the PUSH flag was set in received data.

Berkeley-derived implementations ignore a received PUSH flag because they normally never delay the delivery of received data to the application.

Examples

In Figure 20.1 (p. 276) we see the PUSH flag turned on for all eight data segments (4–6, 9, 11–13, and 15). This is because the client did eight writes of 1024 bytes, and each write emptied the send buffer.

Look again at Figure 20.7 (p. 283). We expect the PUSH flag to be set on segment 12, since that is the final data segment. Why was the PUSH flag set on segment 7, when the

sender knew there were still more bytes to send? The reason is that the size of the sender's send buffer is 4096 bytes, even though we specified a single write of 8192 bytes.

Another point to note in Figure 20.7 concerns the three consecutive ACKs, segments 14, 15, and 16. We saw two consecutive ACKs in Figure 20.3, but that was because the receiver had advertised a window of 0 (stopping the sender) so when the window opened up, another ACK was required, with the nonzero window, to restart the sender. In Figure 20.7, however, the window never reaches 0. Nevertheless, when the size of the window increases by 2048 bytes, another ACK is sent (segments 15 and 16) to provide this window update to the other end. (These two window updates in segments 15 and 16 are not needed, since the FIN has been received from the other end, preventing it from sending any more data.) Many implementations send this window update if the window increases by either two maximum sized segments (2048 bytes in this example, with an MSS of 1024) or 50% of the maximum possible window (2048 bytes in this example, with a maximum window of 4096). We'll see this again in Section 22.3 when we examine the silly window syndrome in detail.

As another example of the PUSH flag, look again at Figure 20.3 (p. 279). The reason we see the flag on for the first four data segments (4–7) is because each one caused a segment to be generated by TCP and passed to the IP layer. But then TCP had to stop, waiting for an ACK to move the 4096-byte window. While waiting for the ACK, TCP takes the final 4096 bytes of data from the application. When the window opens up (segment 9) the sending TCP knows it has four segments that it can send immediately, so it only turns on the PUSH flag for the final segment (13).

20.6 Slow Start

In all the examples we've seen so far in this chapter, the sender starts off by injecting multiple segments into the network, up to the window size advertised by the receiver. While this is OK when the two hosts are on the same LAN, if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate router must queue the packets, and it's possible for that router to run out of space. [Jacobson 1988] shows how this naive approach can reduce the throughput of a TCP connection drastically.

TCP is now required to support an algorithm called *slow start*. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end.

Slow start adds another window to the sender's TCP: the *congestion window*, called *cwnd*. When a new connection is established with a host on another network, the congestion window is initialized to one segment (i.e., the segment size announced by the other end). Each time an ACK is received, the congestion window is increased by one segment. (*cwnd* is maintained in bytes, but slow start always increments it by the segment size.) The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential increase.

At some point the capacity of the internet can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large. When we talk about TCP's timeout and retransmission algorithms in the next chapter, we'll see how this is handled, and what happens to the congestion window. For now, let's watch slow start in action.

An Example

Figure 20.8 shows data being sent from the host `sun` to the host `vangogh.cs.berkeley.edu`. The data traverses a slow SLIP link, which should be the bottleneck. (We have removed the connection establishment from this time line.)

We see the sender transmit one segment with 512 bytes of data and then wait for its ACK. The ACK is received 716 ms later, which is an indicator of the round-trip time. The congestion window is then increased to two segments, and two segments are sent. When the ACK in segment 5 is received, the congestion window is increased to three segments. Although three more could be sent, only two are sent before another ACK is received.

We'll return to slow start in Section 21.6 and see how it's normally implemented with another technique called *congestion avoidance*.

20.7 Bulk Data Throughput

Let's look at the interaction of the window size, the windowed flow control, and slow start on the throughput of a TCP connection carrying bulk data.

Figure 20.9 shows the steps over time of a connection between a sender on the left and a receiver on the right. Sixteen units of time are shown. We show only discrete units of time in this figure, for simplicity. We show segments carrying data going from the left to right in the top half of each picture, numbered 1, 2, 3, and so on. The ACKs go in the other direction in the bottom half of each picture. We draw the ACKs smaller, and show the segment number being acknowledged.

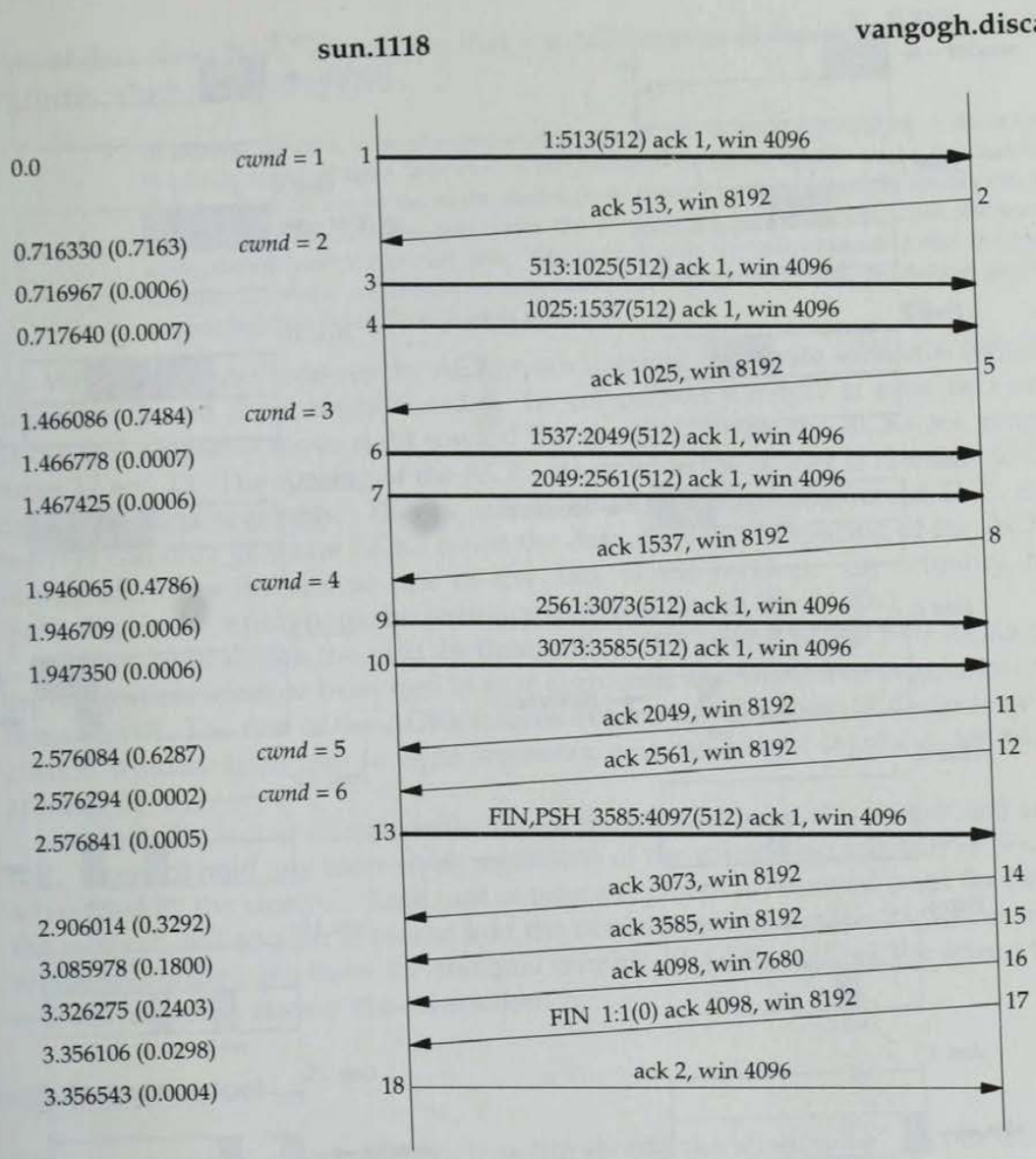


Figure 20.8 Example of slow start.

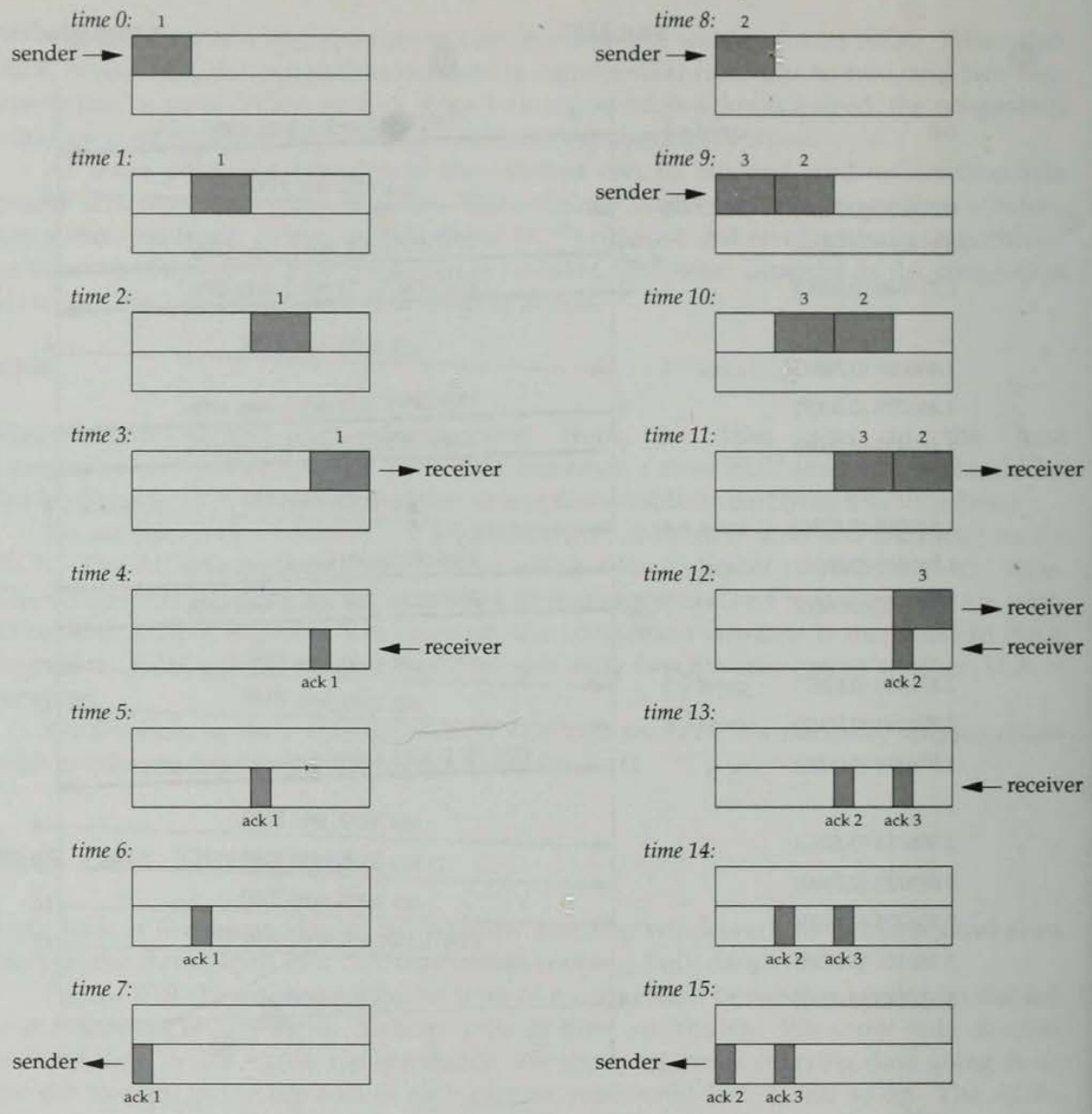


Figure 20.9 Times 0–15 for bulk data throughput example.

At time 0 the sender transmits one segment. Since the sender is in slow start (its congestion window is one segment), it must wait for the acknowledgment of this segment before continuing.

At times 1, 2, and 3 the segment moves one unit of time to the right. At time 4 the receiver reads the segment and generates the acknowledgment. At times 5, 6, and 7 the ACK moves to the left one unit, back to the sender. We have a round-trip time (RTT) of 8 units of time.

We have purposely drawn the ACK segment smaller than the data segment, since it's normally just an IP header and a TCP header. We're showing only a unidirectional

flow of data here. Also, we assume that the ACK moves at the same speed as the data segment, which isn't always true.

> In general the time to send a packet depends on two factors: a propagation delay (caused by the finite speed of light, latencies in transmission equipment, etc.) and a transmission delay that depends on the speed of the media (how many bits per second the media can transmit). For a given path between two nodes the propagation delay is fixed while the transmission delay depends on the packet size. At lower speeds the transmission delay dominates (e.g., Exercise 7.2 where we didn't even consider the propagation delay), whereas at gigabit speeds the propagation delay dominates (e.g., Figure 24.6).

When the sender receives the ACK it can transmit two more segments (which we've numbered 2 and 3), at times 8 and 9. Its congestion window is now two segments. These two segments move right toward the receiver, where the ACKs are generated at times 12 and 13. The spacing of the ACKs returned to the sender is identical to the spacing of the data segments. This is called the *self-clocking* behavior of TCP. Since the receiver can only generate ACKs when the data arrives, the spacing of the ACKs at the sender identifies the arrival rate of the data at the receiver. (In actuality, however, queueing on the return path can change the arrival rate of the ACKs.)

Figure 20.10 shows the next 16 time units. The arrival of the two ACKs increases the congestion window from two to four segments, and these four segments are sent at times 16–19. The first of the ACKs returns at time 23. The four ACKs increase the congestion window from four to eight segments, and these eight segments are transmitted at times 24–31.

At time 31, and at all successive times, the pipe between the sender and receiver is full. It cannot hold any more data, regardless of the congestion window or the window advertised by the receiver. Each unit of time a segment is removed from the network by the receiver, and another is placed into the network by the sender. However many data segments fill the pipe, there are an equal number of ACKs making the return trip. This is the ideal steady state of the connection.

Bandwidth-Delay Product

We can now answer the question: how big should the window be? In our example, the sender needs to have eight segments outstanding and unacknowledged at any time, for maximum throughput. The receiver's advertised window must be that large, since that limits how much the sender can transmit.

We can calculate the capacity of the pipe as

$$\textit{capacity}\ (\text{bits}) = \textit{bandwidth}\ (\text{bits/sec}) \times \textit{round-trip time}\ (\text{sec})$$

This is normally called the *bandwidth-delay product*. This value can vary widely, depending on the network speed and the RTT between the two ends. For example, a T1 telephone line (1,544,000 bits/sec) across the United States (about a 60-ms RTT) gives a bandwidth-delay product of 11,580 bytes. This is reasonable in terms of the buffer sizes we talked about in Section 20.4, but a T3 telephone line (45,000,000 bits/sec) across the United States gives a bandwidth-delay product of 337,500 bytes, which is bigger than the maximum allowable TCP window advertisement (65535 bytes). We describe the

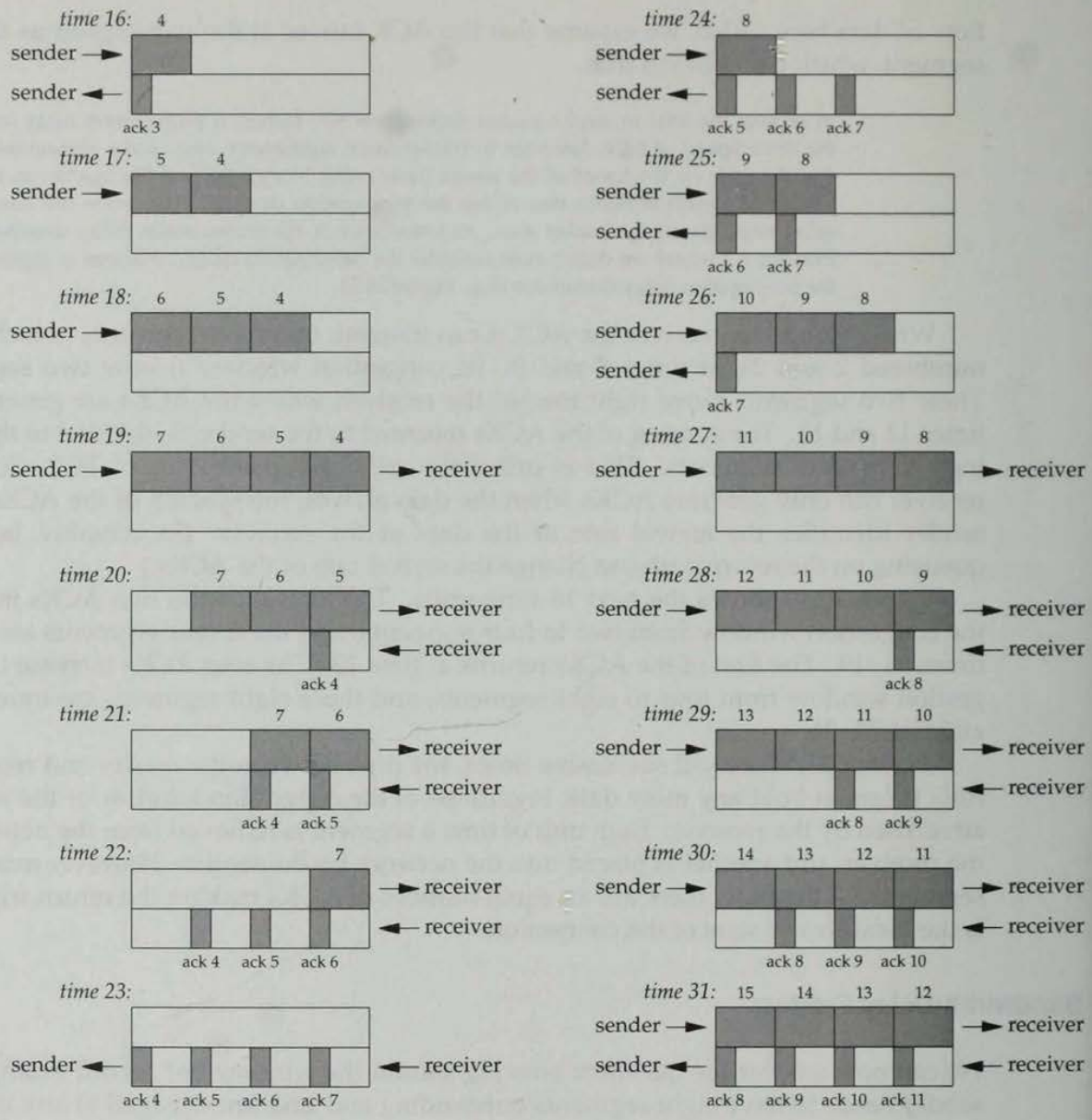


Figure 20.10 Times 16–31 for bulk data throughput example.

new TCP window scale option in Section 24.4 that gets around this current limitation of TCP.

> The value 1,544,000 bits/sec for a T1 phone line is the raw bit rate. The data rate is actually 1,536,000 bits/sec, since 1 bit in 193 is used for framing. The raw bit rate of a T3 phone line is actually 44,736,000 bits/sec, and the data rate can reach 44,210,000 bits/sec. For our discussion we'll use 1.544 Mbits/sec and 45 Mbits/sec.

Either the bandwidth or the delay can affect the capacity of the pipe between the sender and receiver. In Figure 20.11 we show graphically how a doubling of the RTT doubles the capacity of the pipe.

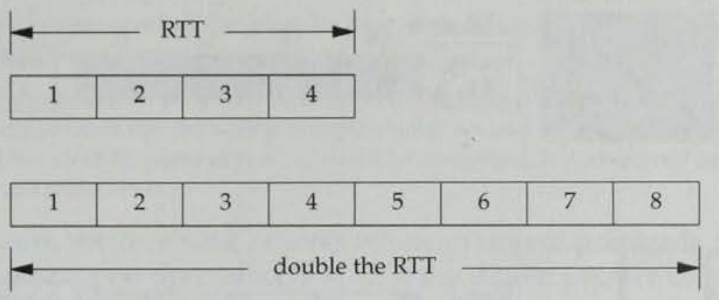


Figure 20.11 Doubling the RTT doubles the capacity of the pipe.

In the lower illustration of Figure 20.11, with the longer RTT, the pipe can hold eight segments, instead of four.

Similarly, Figure 20.12 shows that doubling the bandwidth also doubles the capacity of the pipe.

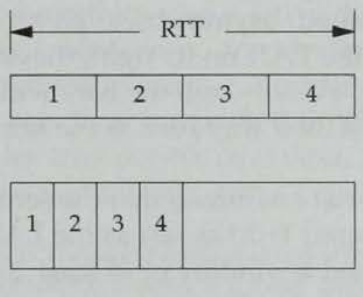


Figure 20.12 Doubling the bandwidth doubles the capacity of the pipe.

In the lower illustration of Figure 20.12, we assume that the network speed has doubled, allowing us to send four segments in half the time as in the top picture. Again, the capacity of the pipe has doubled. (We assume that the segments in the top half of this figure have the same area, that is the same number of bits, as the segments in the bottom half.)

Congestion

Congestion can occur when data arrives on a big pipe (a fast LAN) and gets sent out a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs.

Figure 20.13 shows a typical scenario with a big pipe feeding a smaller pipe. We say this is typical because most hosts are connected to LANs, with an attached router that is connected to a slower WAN. (Again, we are assuming the areas of all the data segments (9–20) in the top half of the figure are all the same, and the areas of all the acknowledgments in the bottom half are all the same.)

In this figure we have labeled the router R1 as the "bottleneck," because it is the congestion point. It can receive packets from the LAN on its left faster than they can be

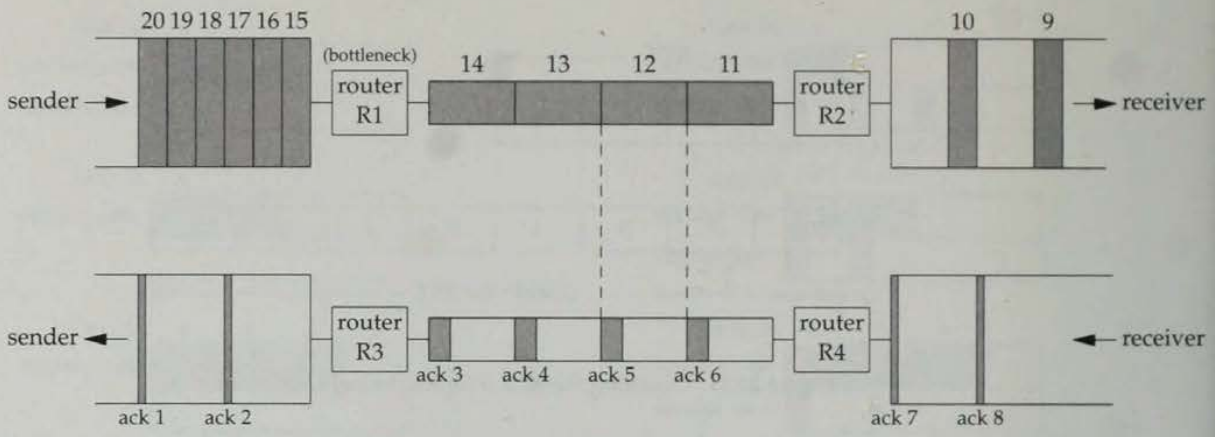


Figure 20.13 Congestion caused by a bigger pipe feeding a smaller pipe.

sent out the WAN on its right. (Commonly R1 and R3 are the same router, as are R2 and R4, but that's not required; asymmetrical paths can occur.) When router R2 puts the received packets onto the LAN on its right, they maintain the same spacing as they did on the WAN on its left, even though the bandwidth of the LAN is higher. Similarly, the spacing of the ACKs on their way back is the same as the spacing of the slowest link in the path.

In Figure 20.13 we have assumed that the sender did not use slow start, and sent the segments we've numbered 1–20 as fast as the LAN could take them. (This assumes the receiving host advertised a window of at least 20 segments.) The spacing of the ACKs will correspond to the bandwidth of the slowest link, as we show. We are assuming the bottleneck router has adequate buffering for all 20 segments. This is not guaranteed, and can lead to that router discarding packets. We'll see how to avoid this when we talk about congestion avoidance in Section 21.6.

20.8 Urgent Mode

TCP provides what it calls *urgent mode*, allowing one end to tell the other end that "urgent data" of some form has been placed into the normal stream of data. The other end is notified that this urgent data has been placed into the data stream, and it's up to the receiving end to decide what to do.

The notification from one end to the other that urgent data exists in the data stream is done by setting two fields in the TCP header (Figure 17.2, p. 225). The URG bit is turned on and the 16-bit *urgent pointer* is set to a positive offset that must be added to the sequence number field in the TCP header to obtain the sequence number of the last byte of urgent data.

There is continuing debate about whether the urgent pointer points to the last byte of urgent data, or to the byte following the last byte of urgent data. The original TCP specification gave

both interpretations but the Host Requirements RFC identifies which is correct: the urgent pointer points to the last byte of urgent data.

The problem, however, is that most implementations (i.e., the Berkeley-derived implementations) continue to use the wrong interpretation. An implementation that follows the specification in the Host Requirements RFC might be compliant, but might not communicate correctly with most other hosts.

TCP must inform the receiving process when an urgent pointer is received and one was not already pending on the connection, or if the urgent pointer advances in the data stream. The receiving application can then read the data stream and must be able to tell when the urgent pointer is encountered. As long as data exists from the receiver's current read position until the urgent pointer, the application is considered to be in an "urgent mode." After the urgent pointer is passed, the application returns to its normal mode.

TCP itself says little more about urgent data. There is no way to specify where the urgent data starts in the data stream. The only information sent across the connection by TCP is that urgent mode has begun (the URG bit in the TCP header) and the pointer to the last byte of urgent data. Everything else is left to the application.

Unfortunately many implementations incorrectly call TCP's urgent mode *out-of-band* data. If an application really wants a separate out-of-band channel, a second TCP connection is the easiest way to accomplish this. (Some transport layers do provide what most people consider true out-of-band data: a logically separate data path using the same connection as the normal data path. This is not what TCP provides.)

The confusion between TCP's urgent mode and out-of-band data is also because the predominant programming interface, the sockets API, maps TCP's urgent mode into what sockets calls out-of-band data.

What is urgent mode used for? The two most commonly used applications are Telnet and Rlogin, when the interactive user types the interrupt key, and we show examples of this use of urgent mode in Chapter 26. Another is FTP, when the interactive user aborts a file transfer, and we show an example of this in Chapter 27.

Telnet and Rlogin use urgent mode from the server to the client because it's possible for this direction of data flow to be stopped by the client TCP (i.e., it advertises a window of 0). But if the server process enters urgent mode, the server TCP immediately sends the urgent pointer and the URG flag, even though it can't send any data. When the client TCP receives this notification, it in turn notifies the client process, so the client can read its input from the server, to open the window, and let the data flow.

What happens if the sender enters urgent mode multiple times before the receiver processes all the data up through the first urgent pointer? The urgent pointer just advances in the data stream, and its previous position at the receiver is lost. There is only one urgent pointer at the receiver and its value is overwritten when a new value for the urgent pointer arrives from the other end. This means if the contents of the data stream that are written by the sender when it enters urgent mode are important to the receiver, these data bytes must be specially marked (somehow) by the sender. We'll see that Telnet marks all of its command bytes in the data stream by prefixing them with a byte of 255.

An Example

Let's watch how TCP sends urgent data, even when the receiver's window is closed. We'll start our `sock` program on the host `bsdi` and have it pause for 10 seconds after the connection is established (the `-P` option), before it reads from the network. This lets the other end fill the send window.

```
bsdi % sock -i -s -P10 5555
```

We then start the client on the host `sun` telling it to use a send buffer of 8192 bytes (`-S` option) and perform six 1024-byte writes to the network (`-n` option). We also specify `-U5` telling it to write 1 byte of data and enter urgent mode before writing the fifth buffer to the network. We specify the verbose flag to see the order of the writes:

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
wrote 1024 bytes
wrote 1024 bytes
```

We set the send buffer size to 8192 bytes, to let the sending application immediately write all of its data. Figure 20.14 shows the `tcpdump` output for this exchange. (We have removed the connection establishment.) Lines 1–5 show the sender filling the receiver's window with four 1024-byte segments. The sender is then stopped because the receiver's window is full. (The ACK on line 4 acknowledges data, but does not move the right edge of the window.)

After the fourth application write of normal data, the application writes 1 byte of data and enters urgent mode. Line 6 is the result of this application write. The urgent pointer is set to 4098. The urgent pointer is sent with the URG flag even though the sender cannot send any data.

Five of these ACKs are sent in about 13 ms (lines 6–10). The first is sent when the application writes 1 byte and enters urgent mode. The next two are sent when the application does the final two writes of 1024 bytes. (Even though TCP can't send these 2048 bytes of data, each time the application performs a write, the TCP output function is called, and when it sees that urgent mode has been entered, sends another urgent notification.) The fourth of these ACKs occurs when the application closes its end of the connection. (The TCP output function is again called.) The sending application terminates milliseconds after it starts—before the receiving application has issued its first write. TCP queues all the data and sends it when it can. (This is why we specified a send buffer size of 8192—so all the data can fit in the buffer.) The fifth of these ACKs is probably generated by the reception of the ACK on line 4. The sending TCP has probably already queued its fourth segment for output (line 5) before this ACK arrives. The receipt of this ACK from the other end also causes the TCP output routine to be called.

```
 1  0.0                  sun.1305 > bsdi.5555: P 1:1025(1024) ack 1 win 4096
 2  0.073743 (0.0737)    sun.1305 > bsdi.5555: P 1025:2049(1024) ack 1 win 4096
 3  0.096969 (0.0232)    sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
 4  0.157514 (0.0605)    bsdi.5555 > sun.1305: . ack 3073 win 1024
 5  0.164267 (0.0068)    sun.1305 > bsdi.5555: P 3073:4097(1024) ack 1 win 4096

 6  0.167961 (0.0037)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 7  0.171969 (0.0040)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 8  0.176196 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 9  0.180373 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
10  0.180768 (0.0004)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

11  0.367533 (0.1868)    bsdi.5555 > sun.1305: . ack 4097 win 0
12  0.368478 (0.0009)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

13  9.829712 (9.4612)    bsdi.5555 > sun.1305: . ack 4097 win 2048
14  9.831578 (0.0019)    sun.1305 > bsdi.5555: . 4097:5121(1024) ack 1 win 4096
                                                 urg 4098
15  9.833303 (0.0017)    sun.1305 > bsdi.5555: . 5121:6145(1024) ack 1 win 4096

16  9.835089 (0.0018)    bsdi.5555 > sun.1305: . ack 4097 win 4096
17  9.835913 (0.0008)    sun.1305 > bsdi.5555: FP 6145:6146(1) ack 1 win 4096
18  9.840264 (0.0044)    bsdi.5555 > sun.1305: . ack 6147 win 2048
19  9.842386 (0.0021)    bsdi.5555 > sun.1305: . ack 6147 win 4096
20  9.843622 (0.0012)    bsdi.5555 > sun.1305: F 1:1(0) ack 6147 win 4096
21  9.844320 (0.0007)    sun.1305 > bsdi.5555: . ack 2 win 4096
```

Figure 20.14 `tcpdump` output for TCP urgent mode.

The receiver then acknowledges the final 1024 bytes of data (line 11) but also advertises a window of 0. The sender responds with another segment containing the urgent notification.

The receiver advertises a window of 2048 bytes in line 13, when the application wakes up and reads some of the data from the receive buffer. The next two 1024-byte segments are sent (lines 14 and 15). The first segment has the urgent notification set, since the urgent pointer is within this segment. The second segment has turned the urgent notification off.

When the receiver opens the window again (line 16) the sender transmits the final byte of data (numbered 6145) and also initiates the normal connection termination.

Figure 20.15 shows the sequence numbers of the 6145 bytes of data that are sent. We see that the sequence number of the byte written when urgent mode was entered is 4097, but the value of the urgent pointer in Figure 20.14 is 4098. This confirms that this implementation (SunOS 4.1.3) sets the urgent pointer to 1 byte beyond the last byte of urgent data.

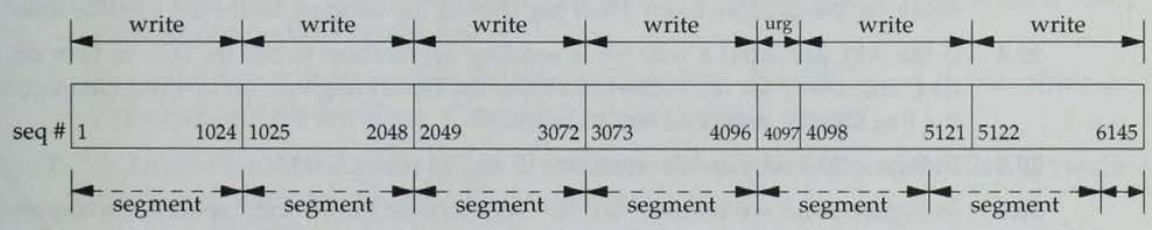


Figure 20.15 Application writes and TCP segments for urgent mode example.

This figure also lets us see how TCP repacketizes the data that the application wrote. The single byte that was output when urgent mode was entered is sent along with the next 1023 bytes of data in the buffer. The next segment also contains 1024 bytes of data, and the final segment contains 1 byte of data.

20.9 Summary

As we said early in the chapter, there is no single way to exchange bulk data using TCP. It is a dynamic process that depends on many factors, some of which we can control (e.g., send and receive buffer sizes) and some of which we have no control over (e.g., network congestion, implementation features). In this chapter we've examined many TCP transfers, explaining all the characteristics and algorithms that we could see.

Fundamental to the efficient transfer of bulk data is TCP's sliding window protocol. We then looked at what it takes for TCP to get the fastest transfer possible by keeping the pipe between the sender and receiver full. We measured the capacity of this pipe as the bandwidth-delay product, and saw the relationship between this and the window size. We return to this concept in Section 24.8 when we look at TCP performance.

We also looked at TCP's PUSH flag, since we'll always see it in trace output, but we have no control over its setting. The final topic was TCP's urgent data, which is often mistakenly called "out-of-band data." TCP's urgent mode is just a notification from the sender to the receiver that urgent data has been sent, along with the sequence number of the final byte of urgent data. The programming interface for the application to use with urgent data is often less than optimal, which leads to much confusion.

Exercises

20.1 In Figure 20.6 (p. 281) we could have shown a byte numbered 0 and a byte numbered 8193. What do these 2 bytes designate?

20.2 Look ahead to Figure 22.1 (p. 324) and explain the setting of the PUSH flag by the host `bsdi`.

20.3 In a Usenet posting someone complained about a throughput of 120,000 bits/sec on a 256,000 bits/sec link with a 128-ms delay between the United States and Japan (47% utilization), and a throughput of 33,000 bits/sec when the link was routed over a satellite (13% utilization). What does the window size appear to be for both cases? (Assume a 500-ms delay for the satellite link.) How big should the window be for the satellite link?

20.4 If the API provided a way for a sending application to tell its TCP to turn on the PUSH flag, and a way for the receiver to tell if the PUSH flag was on in a received segment, could the flag then be used as a record marker?

20.5 In Figure 20.3 why aren't segments 15 and 16 combined?

20.6 In Figure 20.13 we assume that the ACKs come back nicely spaced, corresponding to the spacing of the data segments. What happens if the ACKs are queued somewhere on the return path, causing a bunch of them to arrive at the same time at the sender?

21

TCP Timeout and Retransmission

21.1 Introduction

TCP provides a reliable transport layer. One of the ways it provides reliability is for each end to acknowledge the data it receives from the other end. But data segments and acknowledgments can get lost. TCP handles this by setting a timeout when it sends data, and if the data isn't acknowledged when the timeout expires, it retransmits the data. A critical element of any implementation is the timeout and retransmission strategy. How is the timeout interval determined, and how frequently does a retransmission occur?

We've already seen two examples of timeout and retransmission: (1) In the ICMP port unreachable example in Section 6.5 we saw the TFTP client using UDP employing a simple (and poor) timeout and retransmission strategy: it assumed 5 seconds was an adequate timeout period and retransmitted every 5 seconds. (2) In the ARP example to a nonexistent host (Section 4.5), we saw that when TCP tried to establish the connection it retransmitted its SYN using a longer delay between each retransmission.

TCP manages four different timers for each connection.

1. A *retransmission* timer is used when expecting an acknowledgment from the other end. This chapter looks at this timer in detail, along with related issues such as congestion avoidance.
2. A *persist* timer keeps window size information flowing even if the other end closes its receive window. Chapter 22 describes this timer.
3. A *keepalive* timer detects when the other end on an otherwise idle connection crashes or reboots. Chapter 23 describes this timer.
4. A *2MSL* timer measures the time a connection has been in the TIME_WAIT state. We described this state in Section 18.6.

In this chapter we start with a simple example of TCP's timeout and retransmission and then move to a larger example that lets us look at all the details involved in TCP's timer management. We look at how typical implementations measure the round-trip time of TCP segments and how TCP uses these measurements to estimate the retransmission timeout of the next segment it transmits. We then look at TCP's congestion avoidance—what TCP does when packets are lost—and follow through an actual example where packets are lost. We also look at the newer fast retransmit and fast recovery algorithm, and see how it lets TCP detect lost packets faster than waiting for a timer to expire.

21.2 Simple Timeout and Retransmission Example

Let's first look at the retransmission strategy used by TCP. We'll establish a connection, send some data to verify that everything is OK, disconnect the cable, send some more data, and watch what TCP does:

```
bsdi % telnet svr4 discard
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hello, world                          send this line normally
and hi                                disconnect cable before sending this line
Connection closed by foreign host.    output when TCP gives up after 9 minutes
```

Figure 21.1 shows the `tcpdump` output. (We have removed all the type-of-service information that is set by `bsdi`.)

```
1    0.0                bsdi.1029 > svr4.discard: S 1747921409:1747921409(0)
                                                    win 4096 <mss 1024>
2    0.004811 ( 0.0048) svr4.discard > bsdi.1029: S 3416685569:3416685569(0)
                                                    ack 1747921410
                                                    win 4096 <mss 1024>
3    0.006441 ( 0.0016) bsdi.1029 > svr4.discard: . ack 1 win 4096

4    6.102290 ( 6.0958) bsdi.1029 > svr4.discard: P 1:15(14) ack 1 win 4096
5    6.259410 ( 0.1571) svr4.discard > bsdi.1029: . ack 15 win 4096

6   24.480158 (18.2207) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
7   25.493733 ( 1.0136) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
8   28.493795 ( 3.0001) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
9   34.493971 ( 6.0002) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
10  46.484427 (11.9905) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
11  70.485105 (24.0007) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
12 118.486408 (48.0013) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
13 182.488164 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
14 246.489921 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
15 310.491678 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
16 374.493431 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
17 438.495196 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
18 502.486941 (63.9917) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
19 566.488478 (64.0015) bsdi.1029 > svr4.discard: R 23:23(0) ack 1 win 4096
```

Figure 21.1 Simple example of TCP's timeout and retransmission.

Lines 1, 2, and 3 correspond to the normal TCP connection establishment. Line 4 is the transmission of "hello, world" (12 characters plus the carriage return and linefeed), and line 5 is its acknowledgment. We then disconnect the Ethernet cable from `svr4`.

Line 6 shows "and hi" being sent. Lines 7–18 are 12 retransmissions of that segment, and line 19 is when the sending TCP finally gives up and sends a reset.

Examine the time difference between successive retransmissions: with rounding they occur 1, 3, 6, 12, 24, 48, and then 64 seconds apart. We'll see later in this chapter that the first timeout is actually set for 1.5 seconds after the first transmission. (The reason it occurs 1.0136 seconds after the first transmission, and not exactly 1.5 seconds, was explained in Figure 18.7.) After this the timeout value is doubled for each retransmission, with an upper limit of 64 seconds.

This doubling is called an *exponential backoff*. Compare this to the TFTP example in Section 6.5, where every retransmission occurred 5 seconds after the previous.

The time difference between the first transmission of the packet (line 6 at time 24.480) and the reset (line 19 at time 566.488) is about 9 minutes. Modern TCP's are persistent when trying to send data!

> On most implementations this total timeout value is not tunable. Solaris 2.2 allows the administrator to change this (the `tcp_ip_abort_interval` variable in Section E.4) and its default is only 2 minutes, not the more common 9 minutes.

21.3 Round-Trip Time Measurement

Fundamental to TCP's timeout and retransmission is the measurement of the round-trip time (RTT) experienced on a given connection. We expect this can change over time, as routes might change and as network traffic changes, and TCP should track these changes and modify its timeout accordingly.

First TCP must measure the RTT between sending a byte with a particular sequence number and receiving an acknowledgment that covers that sequence number. Recall from the previous chapter that normally there is not a one-to-one correspondence between data segments and ACKs. In Figure 20.1 (p. 276) this means that one RTT that can be measured by the sender is the time between the transmission of segment 4 (data bytes 1–1024) and the reception of segment 7 (the ACK of bytes 1–2048), even though this ACK is for an additional 1024 bytes. We'll use M to denote the measured RTT.

The original TCP specification had TCP update a smoothed RTT estimator (called R) using the low-pass filter

$$R \leftarrow \alpha R + (1 - \alpha)M$$

where α is a smoothing factor with a recommended value of 0.9. This smoothed RTT is updated every time a new measurement is made. Ninety percent of each new estimate is from the previous estimate and 10% is from the new measurement.

Given this smoothed estimator, which changes as the RTT changes, RFC 793 recommended the retransmission timeout value (RTO) be set to

$$RTO = R\beta$$

where β is a delay variance factor with a recommended value of 2.

[Jacobson 1988] details the problems with this approach, basically that it can't keep up with wide fluctuations in the RTT, causing unnecessary retransmissions. As Jacobson notes, unnecessary retransmissions add to the network load, when the network is already loaded. It is the network equivalent of pouring gasoline on a fire.

What's needed is to keep track of the variance in the RTT measurements, in addition to the smoothed RTT estimator. Calculating the *RTO* based on both the mean and variance provides much better response to wide fluctuations in the round-trip times, than just calculating the *RTO* as a constant multiple of the mean. Figures 5 and 6 in [Jacobson 1988] show a comparison of the RFC 793 *RTO* values for some actual round-trip times, versus the *RTO* calculations we show below, which take into account the variance of the round-trip times.

As described by Jacobson, the mean deviation is a good approximation to the standard deviation, but easier to compute. (Calculating the standard deviation requires a square root.) This leads to the following equations that are applied to each RTT measurement M.

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h(|Err| - D)$$

$$RTO = A + 4D$$

where A is the smoothed RTT (an estimator of the average) and D is the smoothed mean deviation. *Err* is the difference between the measured value just obtained and the current RTT estimator. Both A and D are used to calculate the next retransmission timeout (*RTO*). The gain g is for the average and is set to 1/8 (0.125). The gain for the deviation is h and is set to 0.25. The larger gain for the deviation makes the *RTO* go up faster when the RTT changes.

> [Jacobson 1988] specified $2D$ in the calculation of *RTO*, but after further research, [Jacobson 1990c] changed the value to $4D$, which is what appeared in the BSD Net/1 implementation.

Jacobson specifies a way to do all these calculations using integer arithmetic, and this is the implementation typically used. (That's one reason g, h, and the multiplier 4 are all powers of 2, so the operations can be done using shifts instead of multiplies and divides.)

Comparing the original method with Jacobson's, we see that the calculations of the smoothed average are similar (α is one minus the gain g) but a different gain is used. Also, Jacobson's calculation of the *RTO* depends on both the smoothed RTT and the smoothed mean deviation, whereas the original method used a multiple of the smoothed RTT.

We'll see how these estimators are initialized in the next section, when we go through an example.

Karn's Algorithm

A problem occurs when a packet is retransmitted. Say a packet is transmitted, a timeout occurs, the *RTO* is backed off as shown in Section 21.2, the packet is retransmitted with the longer *RTO*, and an acknowledgment is received. Is the ACK for the first transmission or the second? This is called the *retransmission ambiguity problem.*

[Karn and Partridge 1987] specify that when a timeout and retransmission occur, we cannot update the RTT estimators when the acknowledgment for the retransmitted data finally arrives. This is because we don't know to which transmission the ACK corresponds. (Perhaps the first transmission was delayed and not thrown away, or perhaps the ACK of the first transmission was delayed.)

Also, since the data was retransmitted, and the exponential backoff has been applied to the *RTO*, we reuse this backed off *RTO* for the next transmission. Don't calculate a new *RTO* until an acknowledgment is received for a segment that was not retransmitted.

21.4 An RTT Example

We'll use the following example throughout this chapter to examine various implementation details of TCP's timeout and retransmission, slow start, and congestion avoidance.

Using our `sock` program, 32768 bytes of data are sent from our host `slip` to the discard service on the host `vangogh.cs.berkeley.edu` using the command:

```
slip % sock -D -i -n32 vangogh.cs.berkeley.edu discard
```

From the figure on the inside front cover, `slip` is connected from the 140.252.1 Ethernet by two SLIP links, and from there across the Internet to the destination. With two 9600 bits/sec SLIP links, we expect some measurable delays.

This command performs 32 1024-byte writes, and since the MTU between `slip` and `bsdi` is 296, this becomes 128 segments, each with 256 bytes of user data. The total time for the transfer is about 45 seconds and we see one timeout and three retransmissions.

While this transfer was running we ran `tcpdump` on the host `slip` and captured all the segments sent and received. Additionally we specified the `-D` option to turn on socket debugging (Section A.6). We were then able to run a modified version of the `trpt`(8) program to print numerous variables in the connection control block relating to the round-trip timing, slow start, and congestion avoidance.

Given the volume of trace output, we can't show it all. Instead we'll look at pieces as we proceed through the chapter. Figure 21.2 shows the transfer of data and acknowledgments for the first 5 seconds. We have modified this output slightly from our previous display of `tcpdump` output. Although we only measure the times that the packet is sent or received on the host running `tcpdump`, in this figure we want to show that the packets are crossing in the network (which they are, since this LAN connection is not like a shared Ethernet), and show when the receiving host is probably generating the ACKs. (We have also removed all the window advertisements from this figure. `slip` always advertised a window of 4096, and `vangogh` always advertised a window of 8192.)

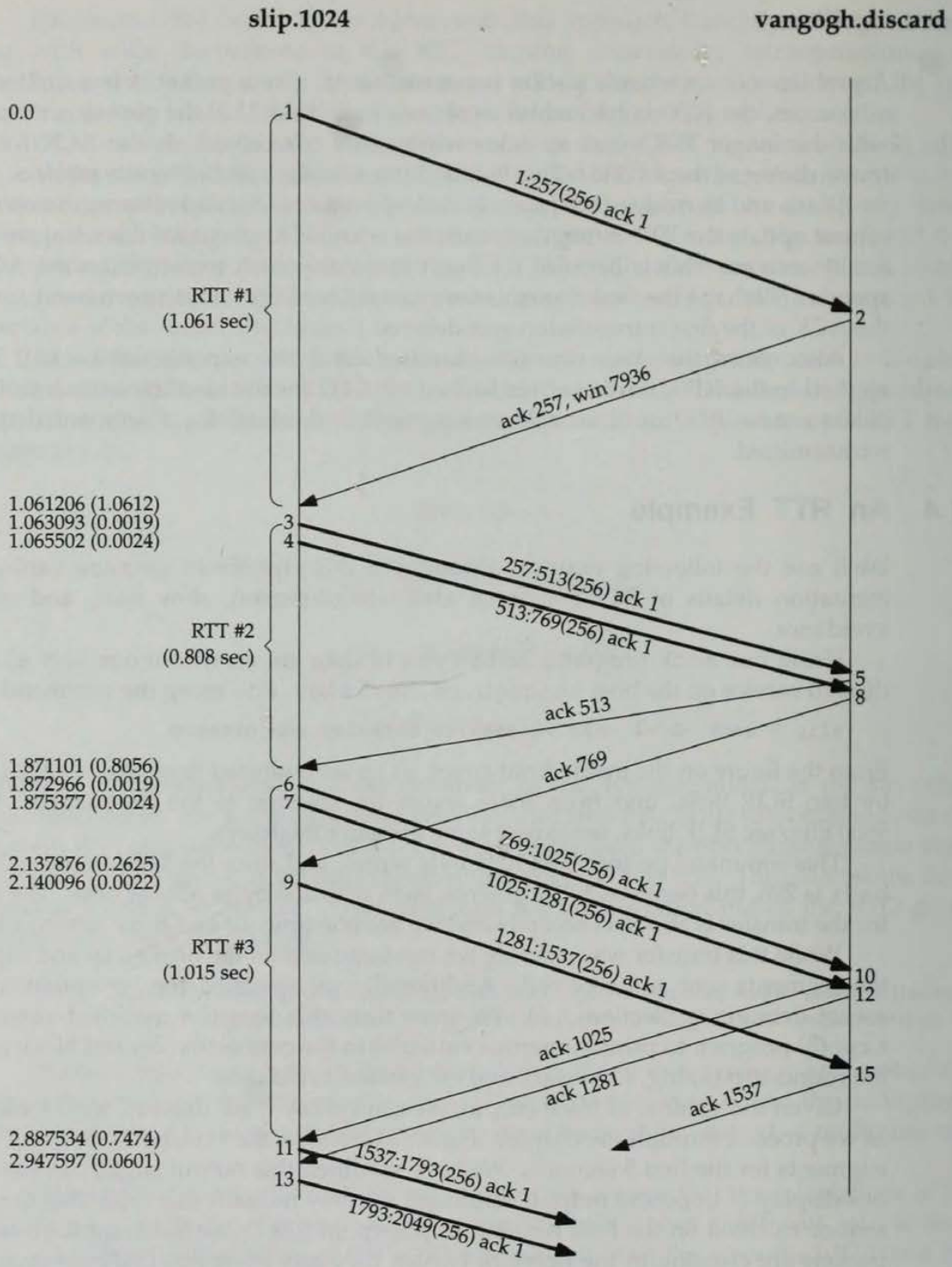


Figure 21.2 Packet exchange and RTT measurement.

Also note in this figure that we have numbered the segments 1–13 and 15, in the order in which they were sent or received on the host `slip`. This correlates with the `tcpdump` output that was collected on this host.

Round-Trip Time Measurements

Three curly braces have been placed on the left side of the time line indicating which segments were timed for RTT calculations. Not all data segments are timed.

Most Berkeley-derived implementations of TCP measure only one RTT value per connection at any time. If the timer for a given connection is already in use when a data segment is transmitted, that segment is not timed.

The timing is done by incrementing a counter every time the 500-ms TCP timer routine is invoked. This means that a segment whose acknowledgment arrives 550 ms after the segment was sent could end up with either a 1 tick RTT (implying 500 ms) or a 2 tick RTT (implying 1000 ms).

In addition to this tick counter for each connection, the starting sequence number of the data in the segment is also remembered. When an acknowledgment that includes this sequence number is received, the timer is turned off. If the data was not retransmitted when the ACK arrives, the smoothed RTT and smoothed mean deviation are updated based on this new measurement.

The timer for the connection in Figure 21.2 is started when segment 1 is transmitted, and turned off when its acknowledgment (segment 2) arrives. Although its RTT is 1.061 seconds (from the `tcpdump` output), examining the socket debug information shows that three of TCP's clock ticks occurred during this period, implying an RTT of 1500 ms.

The next segment timed is number 3. When segment 4 is transmitted 2.4 ms later, it cannot be timed, since the timer for this connection is already in use. When segment 5 arrives, acknowledging the data that was being timed, its RTT is calculated to be 1 tick (500 ms), even though we see that its RTT is 0.808 seconds from the `tcpdump` output.

The timer is started again when segment 6 is transmitted, and turned off when its acknowledgment (segment 10) is received 1.015 seconds later. The measured RTT is 2 clock ticks. Segments 7 and 9 cannot be timed, since the timer is already being used. Also, when segment 8 is received (the ACK of 769), nothing is updated since the acknowledgment was not for bytes being timed.

Figure 21.3 shows the relationship in this example between the actual RTTs that we can determine from the `tcpdump` output, and the counted clock ticks.

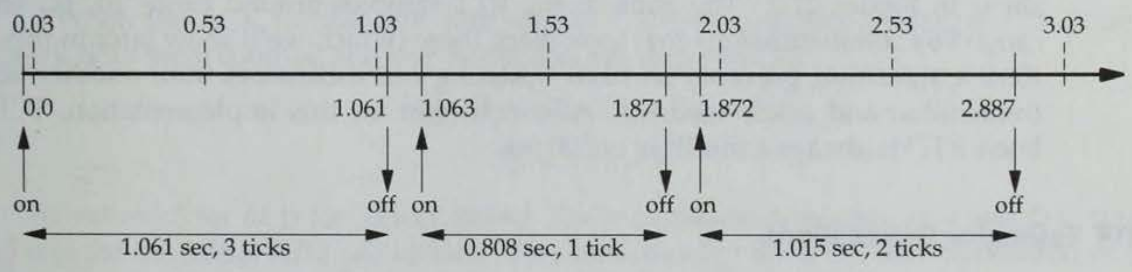


Figure 21.3 RTT measurements and clock ticks.

On the top we show the clock ticks, every 500 ms. On the bottom we show the times output by `tcpdump`, and when the timer for the connection is turned on and off. We know that 3 ticks occur between sending segment 1 and receiving segment 2, 1.061 seconds later, so we assume the first tick occurs at time 0.03. (The first tick must be between 0.00 and 0.061.) The figure then shows how the second measured RTT was counted as 1 tick, and the third as 2 ticks.

In this complete example, 128 segments were transmitted, and 18 RTT samples were collected. Figure 21.4 shows the measured RTT (taken from the `tcpdump` output) along with the *RTO* used by TCP for the timeout (taken from the socket debug output). The x-axis starts at time 0 in Figure 21.2, when the first data segment is transmitted, not when the first SYN is transmitted.

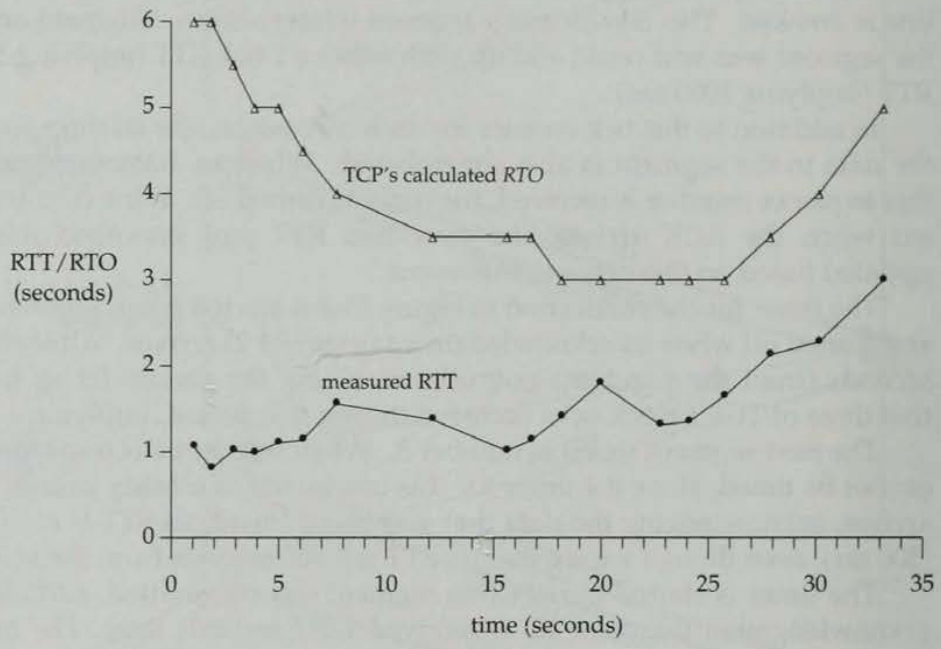


Figure 21.4 Measured RTT and TCP's calculated *RTO* for example.

The first three data points for the measured RTT correspond to the 3 RTTs that we show in Figure 21.2. The gaps in the RTT samples around times 10, 14, and 21 are caused by retransmissions that took place there (which we'll show later in this chapter). Karn's algorithm prevents us from updating our estimators until another segment is transmitted and acknowledged. Also note that for this implementation, TCP's calculated *RTO* is always a multiple of 500 ms.

RTT Estimator Calculations

Let's see how the RTT estimators (the smoothed RTT and the smoothed mean deviation) are initialized and updated, and how each retransmission timeout is calculated.

The variables A and D are initialized to 0 and 3 seconds, respectively. The initial retransmission timeout is calculated using the formula

$$RTO = A + 2D = 0 + 2 \times 3 = 6 \text{ seconds}$$

(The factor $2D$ is used only for this initial calculation. After this $4D$ is added to A to calculate RTO, as shown earlier.) This is the RTO for the transmission of the initial SYN.

It turns out that this initial SYN is lost, and we time out and retransmit. Figure 21.5 shows the first four lines from the `tcpdump` output file.

```
1  0.0                  slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                     win 4096 <mss 256>
2  5.802377 (5.8024)    slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                     win 4096 <mss 256>
3  6.269395 (0.4670)    vangogh.discard > slip.1024: S 1365512705:1365512705(0)
                                                     ack 35648002
                                                     win 8192 <mss 512>
4  6.270796 (0.0014)    slip.1024 > vangogh.discard: . ack 1 win 4096
```

Figure 21.5 Timeout and retransmission of initial SYN.

When the timeout occurs after 5.802 seconds, the current RTO is calculated as

$$RTO = A + 4D = 0 + 4 \times 3 = 12 \text{ seconds}$$

The exponential backoff is then applied to the RTO of 12. Since this is the first timeout we use a multiplier of 2, giving the next timeout value as 24 seconds. The next timeout is calculated using a multiplier of 4, giving a value of 48 seconds: 12×4. (These initial RTOs for the first SYN on a connection, 6 seconds and then 24 seconds, are what we saw in Figure 4.5.)

The ACK arrives 467 ms after the retransmission. The values of A and D are not updated, because of Karn's algorithm dealing with the retransmission ambiguity. The next segment sent is the ACK on line 4, but it is not timed since it is only an ACK. (Only segments containing data are timed.)

When the first data segment is sent (segment 1 in Figure 21.2) the RTO is not changed, again owing to Karn's algorithm. The current value of 24 seconds is reused until an RTT measurement is made. This means the RTO for time 0 in Figure 21.4 is really 24, but we didn't plot that point.

When the ACK for the first data segment arrives (segment 2 in Figure 21.2), three clock ticks were counted and our estimators are initialized as

$$A = M + 0.5 = 1.5 + 0.5 = 2$$

$$D = A/2 = 1$$

(The value 1.5 for M is for 3 clock ticks.) The previous initialization of A and D to 0 and 3 was for the initial RTO calculation. This initialization is for the first calculation of the estimators using the first RTT measurement M. The RTO is calculated as

$$RTO = A + 4D = 2 + 4 \times 1 = 6 \text{ seconds}$$

When the ACK for the second data segment arrives (segment 5 in Figure 21.2), 1 clock tick is counted (0.5 seconds) and our estimators are updated as

$$Err = M - A = 0.5 - 2 = -1.5$$

$$A = A + gErr = 2 - 0.125 \times 1.5 = 1.8125$$

$$D = D + h(|Err| - D) = 1 + 0.25 \times (1.5 - 1) = 1.125$$

$$RTO = A + 4D = 1.8125 + 4 \times 1.125 = 6.3125$$

There are some subtleties in the fixed-point representations of Err, A, and D, and the fixed-point calculations that are actually used (which we've shown in floating-point for simplicity). These differences yield an RTO of 6 seconds (not 6.3125), which is what we plot in Figure 21.4 for time 1.871.

Slow Start

We described the slow start algorithm in Section 20.6. We can see it in action again in Figure 21.2 (p. 302).

Only one segment is initially transmitted on the connection, and its acknowledgment must be received before another segment is transmitted. When segment 2 is received, two more segments are transmitted.

21.5 Congestion Example

Now let's look at the transmission of the data segments. Figure 21.6 is a plot of the starting sequence number in a segment versus the time that the segment was sent. This provides a nice way to visualize the data transmission. Normally the data points should move up and to the right, with the slope of the points being the transfer rate. Retransmissions will appear as motion down and to the right.

At the beginning of Section 21.4 we said the total time for the transfer was about 45 seconds, but we show only 35 seconds in this figure. These 35 seconds account for sending the data segments only. The first data segment was not transmitted until 6.3 seconds after the first SYN was sent, because the first SYN appears to have been lost and was retransmitted (Figure 21.5). Also, after the final data segment and the FIN were sent (at time 34.1 in Figure 21.6) it took another 4.0 seconds to receive the final 14 ACKs from the receiver, before the receiver's FIN was received.

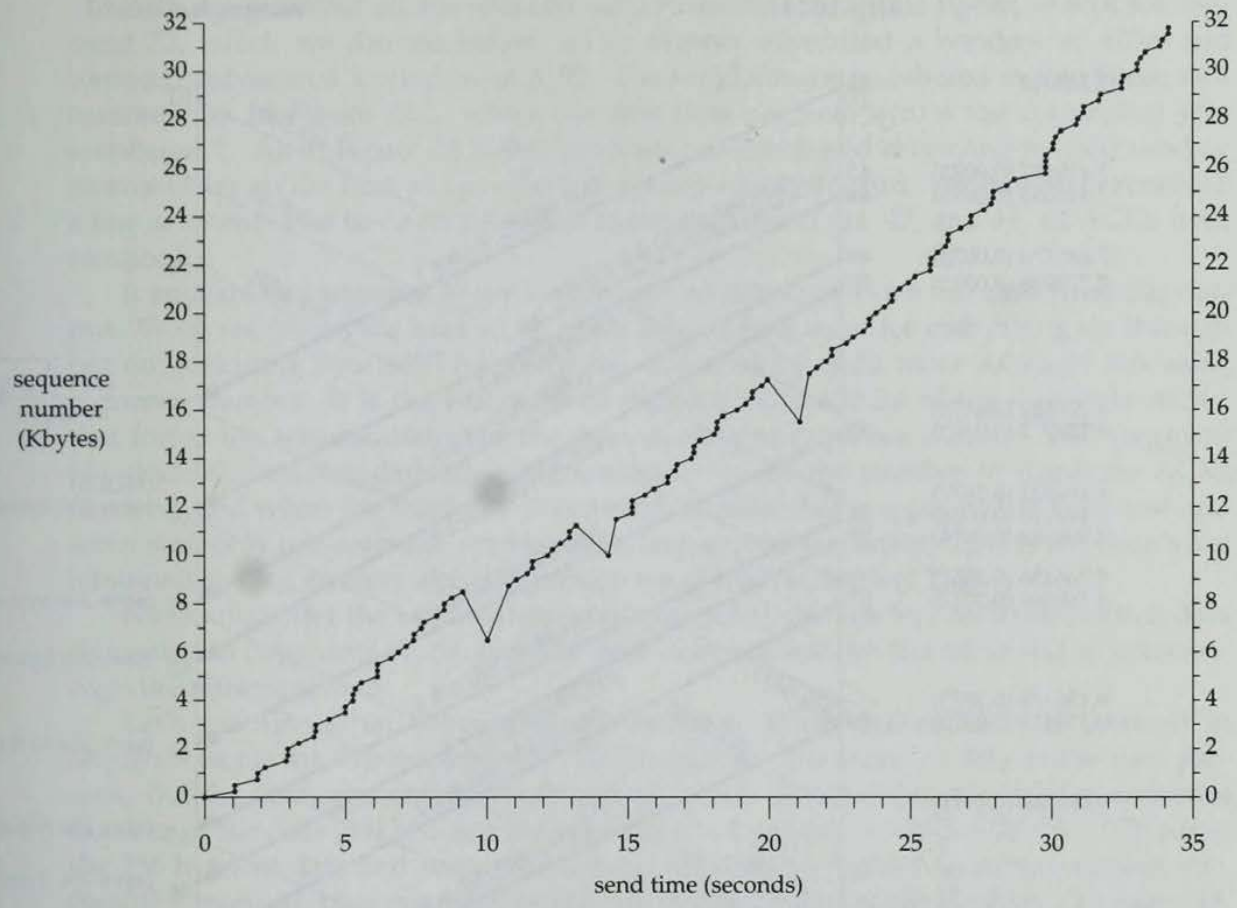


Figure 21.6 Sending of 32768 bytes of data from `slip` to `vangogh`.

We can immediately see the three retransmissions around times 10, 14, and 21 in Figure 21.6. At each of these three points we can also see that only one segment is retransmitted, because only one dot dips below the upward slope.

Let's examine the first of these dips in detail (around the 10-second mark). From the `tcpdump` output we can put together Figure 21.6.

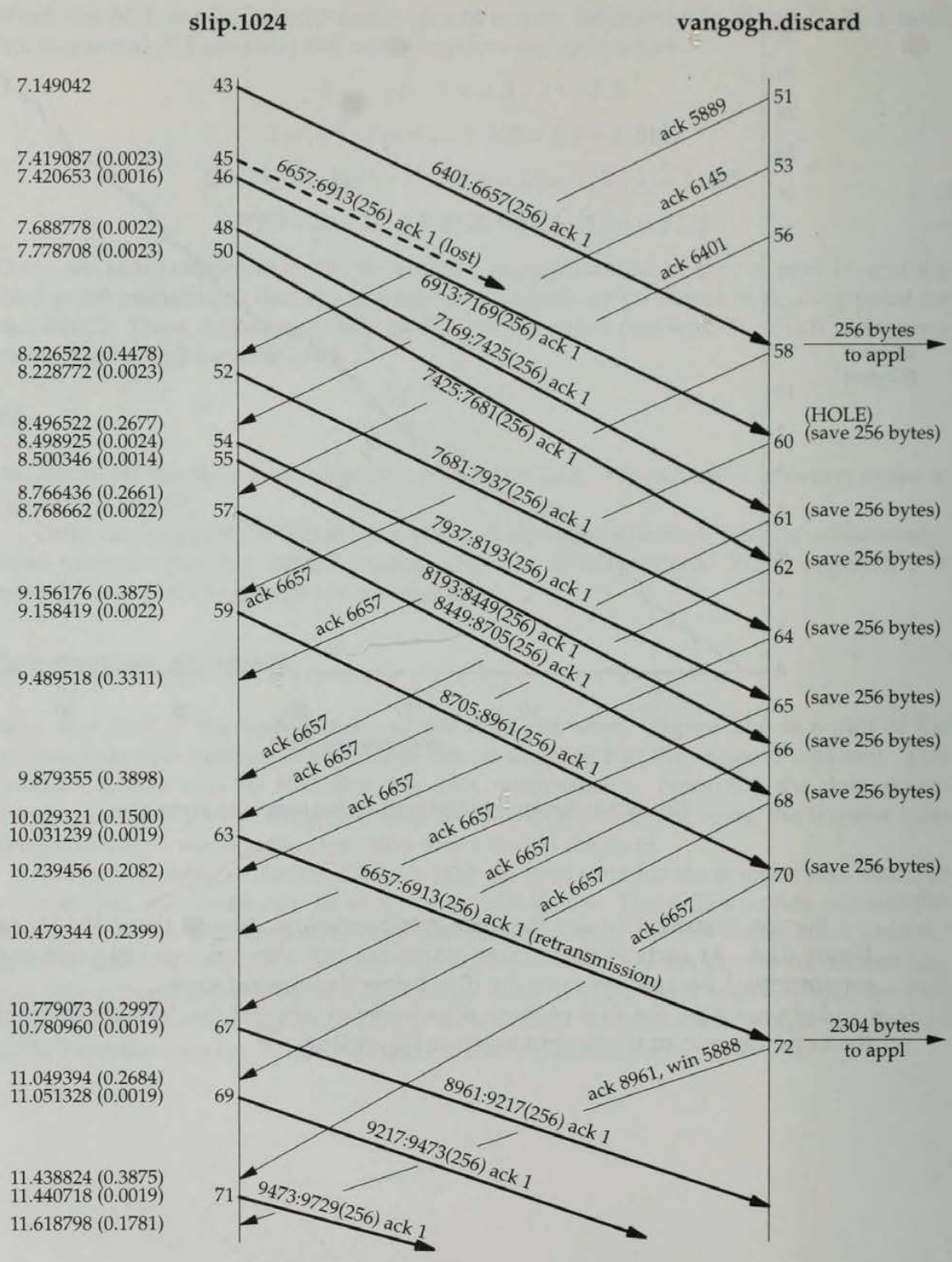


Figure 21.7 Packet exchange for retransmission around the 10-second mark.

We have removed all the window advertisements from this figure, except for segment 72, which we discuss below. `slip` always advertised a window of 4096, and `vangogh` advertised a window of 8192. The segments are numbered in this figure as a continuation of Figure 21.2, where the first data segment across the connection was numbered 1. As in Figure 21.2, the segments are numbered according to their send or receive order on the host `slip`, where `tcpdump` was being run. We have also removed a few segments that have no relevance to the discussion (44, 47, and 49, all ACKs from `vangogh`).

It appears that segment 45 got lost or arrived damaged—we can't tell from this output. What we see on the host `slip` is the acknowledgment for everything up through but not including byte 6657 (segment 58), followed by eight more ACKs of this same sequence number. It is the reception of segment 62, the third of the duplicate ACKs, that forces the retransmission of the data starting at sequence number 6657 (segment 63). Indeed, Berkeley-derived implementations count the number of duplicate ACKs received, and when the third one is received, assume that a segment has been lost and retransmit only one segment, starting with that sequence number. This is Jacobson's *fast retransmit and fast recovery algorithm*, which we discuss in Section 21.7.

Notice that after the retransmission (segment 63), the sender continues normal data transmission (segments 67, 69, and 71). TCP does not wait for the other end to acknowledge the retransmission.

Let's examine what happens at the receiver. When normal data is received in sequence (segment 43), the receiving TCP passes the 256 bytes of data to the user process. But the next segment received (segment 46) is out of order: the starting sequence number of the data (6913) is not the next expected sequence number (6657). TCP saves the 256 bytes of data and responds with an ACK of the highest sequence number successfully received, plus one (6657). The next seven segments received by `vangogh` (48, 50, 52, 54, 55, 57, and 59) are also out of order. The data is saved by the receiving TCP, and duplicate ACKs are generated.

Currently there is no way for TCP to tell the other end that a segment is missing. Also, TCP cannot acknowledge out-of-order data. All `vangogh` can do at this point is continue sending the ACKs of 6657.

When the missing data arrives (segment 63), the receiving TCP now has data bytes 6657–8960 in its buffer, and passes these 2304 bytes to the user process. All 2304 bytes are acknowledged in segment 72. Also notice that this ACK advertises a window of 5888 (8192 – 2304), since the user process hasn't had a chance to read the 2304 bytes that are ready for it.

If we look in detail at the `tcpdump` output for the dips around times 14 and 21 in Figure 21.6, we see that they too were caused by the receipt of three duplicate ACKs, indicating that a packet had been lost. In each of these cases only a single packet was retransmitted.

We'll continue this example in Section 21.8, after describing more about the congestion avoidance algorithms.

21.6 Congestion Avoidance Algorithm

Slow start, which we described in Section 20.6, is the way to initiate data flow across a connection. But at some point we'll reach the limit of an intervening router, and packets can be dropped. Congestion avoidance is a way to deal with lost packets. It is described in [Jacobson 1988].

The assumption of the algorithm is that packet loss caused by damage is very small (much less than 1%), therefore the loss of a packet signals congestion somewhere in the network between the source and destination. There are two indications of packet loss: a timeout occurring and the receipt of duplicate ACKs. (We saw the latter in Section 21.5. If we are using a timeout as an indication of congestion, we can see the need for a good RTT algorithm, such as that described in Section 21.3.)

Congestion avoidance and slow start are independent algorithms with different objectives. But when congestion occurs we want to slow down the transmission rate of packets into the network, and then invoke slow start to get things going again. In practice they are implemented together.

Congestion avoidance and slow start require that two variables be maintained for each connection: a congestion window, *cwnd*, and a slow start threshold size, *ssthresh*. The combined algorithm operates as follows:

1. Initialization for a given connection sets *cwnd* to one segment and *ssthresh* to 65535 bytes.
2. The TCP output routine never sends more than the minimum of *cwnd* and the receiver's advertised window.

   Congestion avoidance is a type of flow control imposed by the sender, while the advertised window is a type of flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion, while the latter is related to the amount of available buffer space at the receiver for this connection.
3. When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), one-half of the current window size (the minimum of *cwnd* and the receiver's advertised window, but at least two segments) is saved in *ssthresh*. Additionally, if the congestion is indicated by a timeout, *cwnd* is set to one segment (i.e., slow start).
4. When new data is acknowledged by the other end, we increase *cwnd*, but the way it increases depends on whether we're performing slow start or congestion avoidance.

   If *cwnd* is less than or equal to *ssthresh*, we're doing slow start; otherwise we're doing congestion avoidance. Slow start continues until we're halfway to where we were when congestion occurred (since we recorded half of the window size that got us into trouble in step 2), and then congestion avoidance takes over.

   Slow start dictates that *cwnd* start at one segment, and be incremented by one segment every time an ACK is received. As mentioned in Section 20.6, this

opens the window exponentially: send one segment, then two, then four, and so on.

Congestion avoidance dictates that *cwnd* be incremented by $1/cwnd$, plus a small fraction of the segment size (the segment size divided by 8), each time an ACK is received. This is an additive increase, compared to slow start's exponential increase. We want to increase *cwnd* by at most one segment each round-trip time (regardless how many ACKs are received in that RTT), whereas slow start will increment *cwnd* by the number of ACKs received in a round-trip time. The addition of the small fraction of the segment size is to allow large windows to open quicker.

The 4.3BSD Tahoe release, described in [Leffler et al. 1989], performed slow start only if the other end was on a different network. This was changed with the 4.3BSD Reno release so that slow start is always performed.

Figure 21.8 is a visual description of slow start and congestion avoidance. We show *cwnd* and *ssthresh* in units of segments, but they're really maintained in bytes.

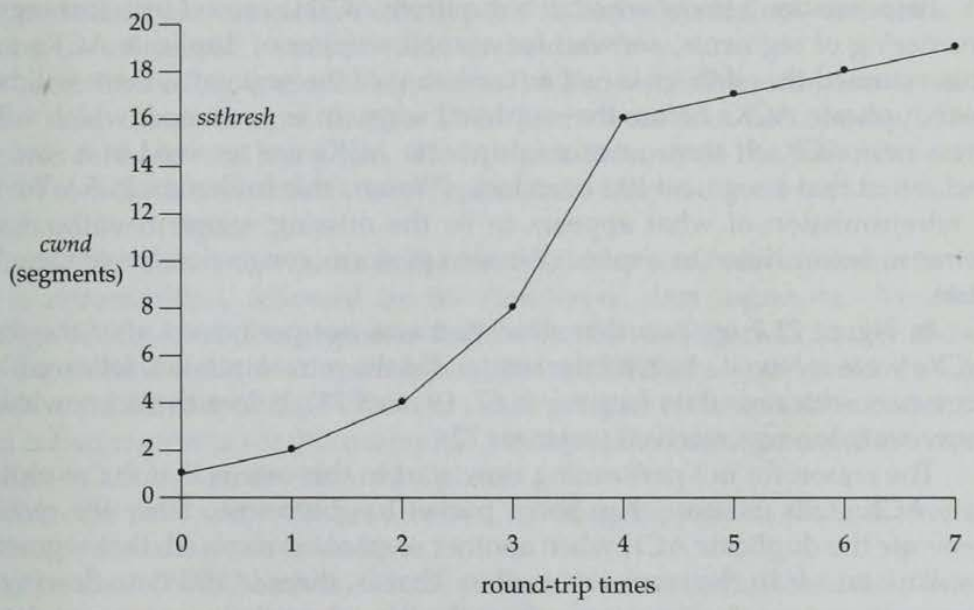


Figure 21.8 Visualization of slow start and congestion avoidance.

In this figure we assume that congestion occurred when *cwnd* had a value of 32 segments. *ssthresh* is then set to 16 segments and *cwnd* is set to 1 segment. One segment is then sent at time 0 and assuming its ACK is returned at time 1, *cwnd* is incremented to 2 segments. Two segments are then sent and assuming their ACKs return by time 2, *cwnd* is incremented to 4 segments (once for each ACK). This exponential increase continues until *cwnd* equals *ssthresh*, after 8 ACKs are received between times 3 and 4. From this point on the increase in *cwnd* is linear, with a maximum increase of one segment per round-trip time.

As we can see in this figure, the term "slow start" is not completely correct. It is a slower transmission of packets than what caused the congestion, but the rate of increase in the number of packets injected into the network increases during slow start. The rate of increase doesn't slow down until *ssthresh* is reached, when congestion avoidance takes over.

21.7 Fast Retransmit and Fast Recovery Algorithm

Modifications to the congestion avoidance algorithm were proposed in 1990 [Jacobson 1990b]. We've already seen these modifications in action in our congestion example (Section 21.5).

Before describing the change, realize that TCP is required to generate an immediate acknowledgment (a duplicate ACK) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected.

Since we don't know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, we wait for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. (We saw this in Section 21.5.) We then perform a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. We also perform congestion avoidance, but not slow start.

In Figure 21.7 we saw that slow start was not performed after the three duplicate ACKs were received. Instead the sender did the retransmission, followed by three more segments with new data (segments 67, 69, and 71), before the acknowledgment of the retransmission was received (segment 72).

The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells us more than just a packet has been lost. Since the receiver can only generate the duplicate ACK when another segment is received, that segment has left the network and is in the receiver's buffer. That is, there is still data flowing between the two ends, and we don't want to reduce the flow abruptly by going into slow start.

This algorithm operates as follows.

1. When the third duplicate ACK is received, set *ssthresh* to one-half the current congestion window, *cwnd*.

   Retransmit the missing segment.

   Set *cwnd* to *ssthresh* plus 3 times the segment size.

2. Each time another duplicate ACK arrives, increment *cwnd* by the segment size and transmit a packet (if allowed by the new value of *cwnd*).

3. When the next ACK arrives that acknowledges new data, set *cwnd* to *ssthresh* (the value set in step 1). This should be the ACK of the retransmission from step 1, one round-trip time after the retransmission. This ACK should also acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK. This step is congestion avoidance, since we're slowing down to one-half the rate we were at when the packet was lost.

We'll see what happens to the two variables *cwnd* and *ssthresh* in the calculations in the next section.

21.8 Congestion Example (Continued)

Watching a connection using `tcpdump` and the socket debug option (which we described in Section 21.4) we can see the values of *cwnd* and *ssthresh* as each segment is transmitted. If the MSS is 256 bytes, the initial values of *cwnd* and *ssthresh* are 256 and 65535, respectively. Each time an ACK is received we can see *cwnd* incremented by the MSS, taking on the values 512, 768, 1024, 1280, and so on. Assuming congestion doesn't occur, eventually the congestion window will exceed the receiver's advertised window, meaning the advertised window will limit the data flow.

A more interesting example is to see what happens when congestion occurs. We'll use the same example from Section 21.4. There were four occurrences of congestion while this example was being run. There was a timeout on the transmission of the initial SYN to establish the connection (Figure 21.5), followed by three lost packets during the data transfer (Figure 21.6).

Figure 21.9 shows the values of the two variables *cwnd* and *ssthresh* when the initial SYN is retransmitted, followed by the first seven data segments. (We showed the exchange of the initial data segments and their ACKs in Figure 21.2.) We show the data bytes transmitted using the `tcpdump` notation: 1:257(256) means bytes 1 through 256.

When the timeout of the SYN occurs, *ssthresh* is set to its minimum value (512 bytes, which is two segments for this example). *cwnd* is set to one segment (256 bytes, which it was already at) to enter the slow start phase.

When the SYN and ACK are received, nothing happens to the two variables, since new data is not being acknowledged.

When the ACK 257 arrives, we are still in slow start since *cwnd* is less than or equal to *ssthresh*, so *cwnd* in incremented by 256. The same thing happens when the ACK 513 arrives.

When the ACK 769 arrives we are no longer in slow start, but enter congestion avoidance. The new value for *cwnd* is calculated as

$$cwnd \leftarrow cwnd + \frac{segsize \times segsize}{cwnd} + \frac{segsize}{8}$$

This is the $1/cwnd$ increase that we mentioned earlier, taking into account that *cwnd* is really maintained in bytes and not segments. For this example we calculate

$$cwnd \leftarrow 768 + \frac{256 \times 256}{768} + \frac{256}{8}$$

Segment# (Figure 21.2)	Action			Variable	
	Send	Receive	Comment	*cwnd*	*ssthresh*
			initialize	256	65535
	SYN				
			timeout	256	512
	SYN		retransmit		
		SYN, ACK			
	ACK				
1	1:257(256)				
2		ACK 257	slow start	512	512
3	257:513(256)				
4	513:769(256)				
5		ACK 513	slow start	768	512
6	769:1025(256)				
7	1025:1281(256)				
8		ACK 769	cong. avoid	885	512
9	1281:1537(256)				
10		ACK 1025	cong. avoid	991	512
11	1537:1793(256)				
12		ACK 1281	cong. avoid	1089	512

Figure 21.9 Example of congestion avoidance.

which equals 885 (using integer arithmetic). When the next ACK 1025 arrives we calculate

$$cwnd \leftarrow 885 + \frac{256 \times 256}{885} + \frac{256}{8}$$

which equals 991.

This additive increase in *cwnd* continues until the first retransmission, around the 10-second mark in Figure 21.6. Figure 21.10 is a plot of the same data as in Figure 21.6, with the value of *cwnd* added.

The first six values for *cwnd* in this figure are the values we calculated for Figure 21.9. It is impossible in this figure to tell the difference visually between the exponential increase during slow start and the additive increase during congestion avoidance, because the slow start phase is so quick.

We need to explain what is happening at the three points where a retransmission occurs. Recall that each of the retransmissions took place because three duplicate ACKs were received, indicating a packet had been lost. This is the fast retransmit and fast recovery algorithm from Section 21.7. *ssthresh* is immediately set to one-half the window size that was in effect when the retransmission took place, but *cwnd* is allowed to keep increasing while the duplicate ACKs are received, since each duplicate ACK means that a segment has left the network (the receiving TCP has buffered it, waiting for the missing hole in the data to arrive).

Figure 21.10 is similar to Figure 21.9, showing the values of *cwnd* and *ssthresh*. The segment numbers in the first column correspond to Figure 21.7.

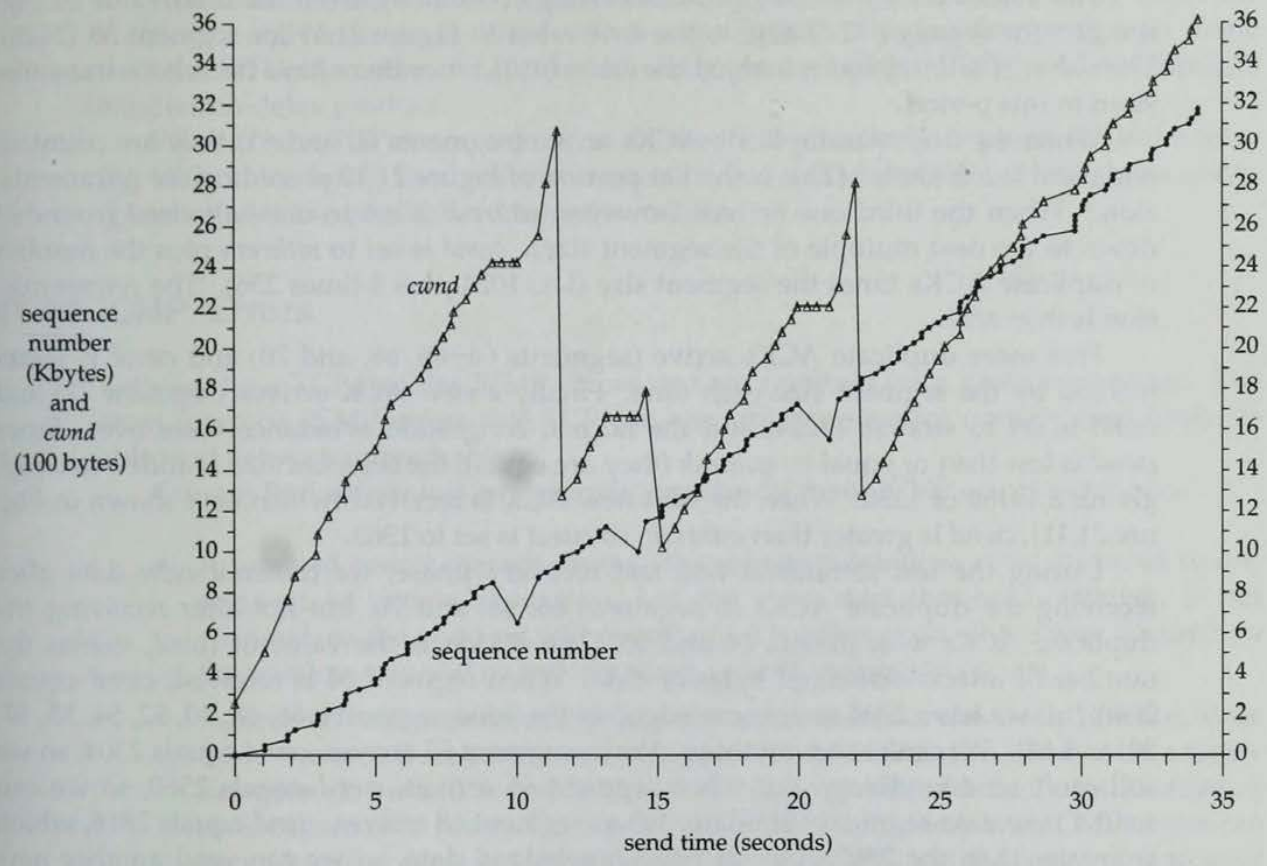


Figure 21.10 Value of *cwnd* and send sequence number while data is being transmitted.

Segment# (Figure 21.7)	Action			Variable	
	Send	Receive	Comment	*cwnd*	*ssthresh*
58		ACK 6657	ACK of new data	2426	512
59	8705:8961(256)				
60		ACK 6657	duplicate ACK #1	2426	512
61		ACK 6657	duplicate ACK #2	2426	512
62		ACK 6657	duplicate ACK #3	1792	1024
63	6657:6913(256)		retransmission		
64		ACK 6657	duplicate ACK #4	2048	1024
65		ACK 6657	duplicate ACK #5	2304	1024
66		ACK 6657	duplicate ACK #6	2560	1024
67	8961:9217(256)				
68		ACK 6657	duplicate ACK #7	2816	1024
69	9217:9473(256)				
70		ACK 6657	duplicate ACK #8	3072	1024
71	9473:9729(256)				
72		ACK 8961	ACK of new data	1280	1024

Figure 21.11 Example of congestion avoidance (continued).

The values for *cwnd* have been increasing continually, from the final value in Figure 21.9 for segment 12 (1089), to the first value in Figure 21.11 for segment 58 (2426). The value of *ssthresh* has remained the same (512), since there have been no retransmissions in this period.

When the first two duplicate ACKs arrive (segments 60 and 61) they are counted, and *cwnd* is left alone. (This is the flat portion of Figure 21.10 preceding the retransmission.) When the third one arrives, however, *ssthresh* is set to one-half *cwnd* (rounded down to the next multiple of the segment size). *cwnd* is set to *ssthresh* plus the number of duplicate ACKs times the segment size (i.e., 1024 plus 3 times 256). The retransmission is then sent.

Five more duplicate ACKs arrive (segments 64–66, 68, and 70) and *cwnd* is incremented by the segment size each time. Finally a new ACK arrives (segment 72) and *cwnd* is set to *ssthresh* (1024) and the normal congestion avoidance takes over. Since *cwnd* is less than or equal to *ssthresh* (they are equal), the segment size is added to *cwnd*, giving a value of 1280. When the next new ACK is received (which isn't shown in Figure 21.11), *cwnd* is greater than *ssthresh*, so *cwnd* is set to 1363.

During the fast retransmit and fast recovery phase, we transmit new data after receiving the duplicate ACKs in segments 66, 68, and 70, but not after receiving the duplicate ACKs in segments 64 and 65. The reason is the value of *cwnd*, versus the number of unacknowledged bytes of data. When segment 64 is received, *cwnd* equals 2048, but we have 2304 unacknowledged bytes (nine segments: 46, 48, 50, 52, 54, 55, 57, 59, and 63). We can't send anything. When segment 65 arrives, *cwnd* equals 2304, so we still can't send anything. But when segment 66 arrives, *cwnd* equals 2560, so we can send a new data segment. Similarly when segment 68 arrives, *cwnd* equals 2816, which is greater than the 2560 bytes of unacknowledged data, so we can send another new data segment. The same scenario happens when segment 70 is received.

When the next retransmission takes place at time 14.3 in Figure 21.10, it is also triggered by the reception of three duplicate ACKs, so we see the same increase in *cwnd* as one more duplicate ACK arrives, followed by a decrease to 1024.

The retransmission at time 21.1 in Figure 21.10 is also triggered by duplicate ACKs. We receive three more duplicates after the retransmission, so we see three additional increases in *cwnd*, followed by a decrease to 1280. For the remainder of the transfer *cwnd* increases linearly to a final value of 3615.

21.9 Per-Route Metrics

Newer TCP implementations maintain many of the metrics that we've described in this chapter in the routing table entry. When a TCP connection is closed, if enough data was sent to obtain meaningful statistics, and if the routing table entry for the destination is not a default route, the following information is saved in the routing table entry, for the next use of the entry: the smoothed RTT, the smoothed mean deviation, and the slow start threshold. The quantity "enough data" is 16 windows of data. This gives 16 RTT samples, which allows the smoothed RTT filter to converge within 5% of the correct value.

Additionally, the `route`(8) command can be used by the administrator to set the metrics for a given route: the three values mentioned in the preceding paragraph, along with the MTU, the outbound bandwidth-delay product (Section 20.7), and the inbound bandwidth-delay product.

When a new TCP connection is established, either actively or passively, if the routing table entry being used for the connection has values for these metrics, the corresponding variable is initialized from the metrics.

21.10 ICMP Errors

Let's see how TCP handles ICMP errors that are returned for a given connection. The most common ICMP errors that TCP can encounter are source quench, host unreachable, and network unreachable.

Current Berkeley-based implementations handle these ICMP errors as follows:

- A received source quench causes the congestion window, *cwnd*, to be set to one segment to initiate slow start, but the slow start threshold, *ssthresh*, is not changed, so the window will open until it's either open all the way (limited by the window size and round-trip time) or until congestion occurs.
- A received host unreachable or network unreachable is effectively ignored, since these two errors are considered transient. It could be that an intermediate router has gone down and it can take the routing protocols a few minutes to stabilize on an alternative route. During this period either of these two ICMP errors can occur, but they must not abort the connection. Instead, TCP keeps trying to send the data that caused the error, although it may eventually time out. (Recall in Figure 21.1 that TCP did not give up for 9 minutes.)

  Current Berkeley-based implementations record that the ICMP error occurred, and if the connection times out, the ICMP error is translated into a more relevant error code than "connection timed out."

  > Earlier BSD implementations incorrectly aborted a connection whenever an ICMP host unreachable or network unreachable was received.

An Example

We can see how an ICMP host unreachable is handled by taking our dialup SLIP link down during the middle of a connection. We establish a connection from the host `slip` to the host `aix`. (From the figure on the inside front cover we see that this connection goes through our dialup SLIP link.) After establishing the connection and transferring some data, the dialup SLIP link between the routers `sun` and `netb` is taken down. This causes the default routing table entry on `sun` (which we showed in Section 9.2) to be removed. We expect `sun` to then respond to IP datagrams destined for the 140.252.1 Ethernet with an ICMP host unreachable. We want to see how TCP handles these ICMP errors.

Here is the interactive session on the host `slip`:

```
slip % sock aix echo                    run our sock program
test line                               type this line
test line                               and it's echoed
                                        SLIP link is brought down here
another line                            then type this line and watch retransmissions
                                        SLIP link is reestablished here
another line                            and the line and its echo are exchanged
line number 3
line number 3
the last line
                                        SLIP link is brought down here, and not reestablished
read error: No route to host            TCP finally gives up
```

Figure 21.12 shows the corresponding `tcpdump` output, captured on the router `bsdi`. (We have removed the connection establishment and all the window advertisements.) We connect to the echo server on the host `aix` and type "test line" (line 1). It is echoed (line 2) and the echo is acknowledged (line 3). We then take down the SLIP link.

We type "another line" (line 3) and expect to see TCP time out and retransmit the message. Indeed, this line is sent six times before a reply is received. Lines 4–13 show the first transmission and the next four retransmissions, each of which generates an ICMP host unreachable from the router `sun`. This is what we expect: the IP datagrams go from `slip` to the router `bsdi` (which has a default route that points to `sun`), and then to `sun`, where the broken link is detected.

While these retransmissions are taking place, the SLIP link is brought back up, and the retransmission on line 14 gets delivered. Line 15 is the echo from `aix`, and line 16 is the acknowledgment of the echo.

This shows that TCP ignores the ICMP host unreachable errors and keeps retransmitting. We can also see the expected exponential backoff in each retransmission timeout: the first appears to be 2.5 seconds, which is then multiplied by 2 (giving 5 seconds), then 4 (10 seconds), then 8 (20 seconds), then 16 (40 seconds).

We then type the third line of input ("line number 3") and see it sent on line 17, echoed on line 18, and the echo acknowledged on line 19.

We now want to see what happens when TCP retransmits and gives up, after receiving the ICMP host unreachable, so we take down the SLIP link again. After taking it down we type "the last line" and see it transmitted 13 times before TCP gives up. (We have deleted lines 30–43 from the output. They are additional retransmissions.)

The thing we notice, however, is the error message printed by our `sock` program when it finally gives up: "No route to host." This corresponds to the Unix error associated with the ICMP host unreachable error (Figure 6.12, p. 82). This shows that TCP saves the ICMP error that it receives on the connection, and when it finally gives up, it prints that error, instead of "Connection timed out."

Finally, notice the different retransmission intervals in lines 22–46, compared to lines 6–14. It appears that TCP updated its estimators when the third line we typed was sent and acknowledged without any retransmissions in lines 17–19. The initial retransmission timeout is now 3 seconds, giving successive values of 6, 12, 24, 48, and then the upper limit of 64.

```
 1      0.0                  slip.1035 > aix.echo: P 1:11(10) ack 1
 2      0.212271 (  0.2123)  aix.echo > slip.1035: P 1:11(10) ack 11
 3      0.310685 (  0.0984)  slip.1035 > aix.echo: . ack 11

                             SLIP link brought down here

 4    174.758100 (174.4474)  slip.1035 > aix.echo: P 11:24(13) ack 11
 5    174.759017 (  0.0009)  sun > slip: icmp: host aix unreachable

 6    177.150439 (  2.3914)  slip.1035 > aix.echo: P 11:24(13) ack 11
 7    177.151271 (  0.0008)  sun > slip: icmp: host aix unreachable

 8    182.150200 (  4.9989)  slip.1035 > aix.echo: P 11:24(13) ack 11
 9    182.151189 (  0.0010)  sun > slip: icmp: host aix unreachable

10    192.149671 (  9.9985)  slip.1035 > aix.echo: P 11:24(13) ack 11
11    192.150608 (  0.0009)  sun > slip: icmp: host aix unreachable

12    212.148783 ( 19.9982)  slip.1035 > aix.echo: P 11:24(13) ack 11
13    212.149786 (  0.0010)  sun > slip: icmp: host aix unreachable

                             SLIP link brought up here

14    252.146774 ( 39.9970)  slip.1035 > aix.echo: P 11:24(13) ack 11
15    252.439257 (  0.2925)  aix.echo > slip.1035: P 11:24(13) ack 24
16    252.505331 (  0.0661)  slip.1035 > aix.echo: . ack 24

17    261.977246 (  9.4719)  slip.1035 > aix.echo: P 24:38(14) ack 24
18    262.158758 (  0.1815)  aix.echo > slip.1035: P 24:38(14) ack 38
19    262.305086 (  0.1463)  slip.1035 > aix.echo: . ack 38

                             SLIP link brought down here

20    458.155330 (195.8502)  slip.1035 > aix.echo: P 38:52(14) ack 38
21    458.156163 (  0.0008)  sun > slip: icmp: host aix unreachable

22    461.136904 (  2.9807)  slip.1035 > aix.echo: P 38:52(14) ack 38
23    461.137826 (  0.0009)  sun > slip: icmp: host aix unreachable

24    467.136461 (  5.9986)  slip.1035 > aix.echo: P 38:52(14) ack 38
25    467.137385 (  0.0009)  sun > slip: icmp: host aix unreachable

26    479.135811 ( 11.9984)  slip.1035 > aix.echo: P 38:52(14) ack 38
27    479.136647 (  0.0008)  sun > slip: icmp: host aix unreachable

28    503.134816 ( 23.9982)  slip.1035 > aix.echo: P 38:52(14) ack 38
29    503.135740 (  0.0009)  sun > slip: icmp: host aix unreachable

                             14 lines of output deleted here

44   1000.219573 ( 64.0959)  slip.1035 > aix.echo: P 38:52(14) ack 38
45   1000.220503 (  0.0009)  sun > slip: icmp: host aix unreachable

46   1064.201281 ( 63.9808)  slip.1035 > aix.echo: R 52:52(0) ack 38
47   1064.202182 (  0.0009)  sun > slip: icmp: host aix unreachable
```

Figure 21.12 TCP handling of received ICMP host unreachable error.

21.11 Repacketization

When TCP times out and retransmits, it does not have to retransmit the identical segment again. Instead, TCP is allowed to perform *repacketization*, sending a bigger segment, which can increase performance. (Naturally, this bigger segment cannot exceed the MSS announced by the other receiver.) This is allowed in the protocol because TCP identifies the data being sent and acknowledged by its byte number, not its segment number.

We can easily see this in action. We use our `sock` program to connect to the discard server and type one line. We then disconnect the Ethernet cable and type a second line. While this second line is being retransmitted, we type a third line. We expect the next retransmission to contain both the second and third lines.

```
bsdi % sock svr4 discard
hello there                          first line gets sent OK
                                     then we disconnect the Ethernet cable
line number 2                        this line gets retransmitted
and 3                                type this line before second line sent OK
                                     then reconnect Ethernet cable
```

Figure 21.13 shows the `tcpdump` output. (We have removed the connection establishment, the connection termination, and all the window advertisements.)

```
1   0.0                 bsdi.1032 > svr4.discard: P 1:13(12) ack 1
2   0.140489 ( 0.1405)  svr4.discard > bsdi.1032: . ack 13

                        Ethernet cable disconnected here

3  26.407696 (26.2672)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1
4  27.639390 ( 1.2317)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1
5  30.639453 ( 3.0001)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1

                        third line typed here

6  36.639653 ( 6.0002)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1
7  48.640131 (12.0005)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1
8  72.640768 (24.0006)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1

                        Ethernet cable reconnected here

9  72.719091 ( 0.0783)  svr4.discard > bsdi.1032: . ack 33
```

Figure 21.13 Repacketization of data by TCP.

Lines 1 and 2 show the first line ("hello there") being sent and its acknowledgment. We then disconnect the Ethernet cable and type "line number 2" (14 bytes, including the newline). These bytes are transmitted on line 3, and then retransmitted on lines 4 and 5.

Before the retransmission on line 6 we type "and 3" (6 bytes, including the newline) and see this retransmission contain 20 bytes: both lines that we typed. When the acknowledgment arrives on line 9, it is for all 20 bytes.

21.12 Summary

This chapter has provided a detailed look at TCP's timeout and retransmission strategy. Our first example was a lost SYN to establish a connection and we saw how an exponential backoff is applied to successive retransmission timeout values.

TCP calculates the round-trip time and then uses these measurements to keep track of a smoothed RTT estimator and a smoothed mean deviation estimator. These two estimators are then used to calculate the next retransmission timeout value. Many implementations only measure a single RTT per window. Karn's algorithm removes the retransmission ambiguity problem by preventing us from measuring the RTT when a packet is lost.

Our detailed example, which included three lost packets, let us see many of TCP's algorithms in action: slow start, congestion avoidance, and fast retransmit and fast recovery. We were also able to hand calculate TCP RTT estimators along with the congestion window and slow-start threshold, and verify the values with the actual values from the trace output.

We finished the chapter by looking at the effect various ICMP errors have on a TCP connection and how TCP is allowed to repacketize its data. We saw how the "soft" ICMP errors don't cause a connection to be terminated, but are remembered so that if the connection terminates abnormally, the soft error can be reported.

Exercises

21.1 In Figure 21.5 the first timeout was calculated as 6 seconds and the next as 12 seconds. If the ACK for the initial SYN had not arrived after the 12-second timeout expired, when would the next timeout occur?

21.2 In the discussion following Figure 21.5 we said that the timeout intervals are calculated as 6, 24, and then 48 seconds, as we saw in Figure 4.5. But if we watch a TCP connection to a nonexistent host from an SVR4 system, the timeout intervals are 6, 12, 24, and 48 seconds. What's going on?

21.3 Compare the performance of TCP's sliding window versus TFTP's stop-and-wait protocol as follows. In this chapter we transferred 32768 bytes in about 35 seconds (Figure 21.6) across a link with an RTT that averaged around 1.5 seconds (Figure 21.4). Calculate how long TFTP would take for the same transfer.

21.4 In Section 21.7 we said that the receipt of a duplicate ACK is caused by a segment being lost or reordered. In Section 21.5 we saw the generation of duplicate ACKs caused by a lost segment. Draw a picture showing that a reordering of segments also generates duplicate ACKs.

21.5 There is a noticeable blip in Figure 21.6 between times 28.8 and 29.8. Is this a retransmission?

21.6 In Section 21.6 we said that the 4.3BSD Tahoe release only performed slow start if the destination was on a different network. How do you think "different network" was determined? (*Hint*: Look at Appendix E.)

21.7 In Section 20.2 we said that TCP normally ACKs every other segment. But in Figure 21.2 we see the receiver ACK every segment. Why?

21.8 Are per-route metrics really useful, given the prevalence of default routes?

22

TCP Persist Timer

22.1 Introduction

We've seen that TCP has the receiver perform flow control by specifying the amount of data it is willing to accept from the sender: the window size. What happens when the window size goes to 0? This effectively stops the sender from transmitting data, until the window becomes nonzero.

We saw this scenario in Figure 20.3 (p. 279). When the sender received segment 9, opening the window that was shut down by segment 8, it immediately started sending data. TCP must handle the case of this acknowledgment that opens the window (segment 9) being lost. Acknowledgments are not reliably transmitted—that is, TCP does not ACK acknowledgments, it only ACKs segments containing data.

If an acknowledgment is lost, we could end up with both sides waiting for the other: the receiver waiting to receive data (since it provided the sender with a nonzero window) and the sender waiting to receive the window update allowing it to send. To prevent this form of deadlock from occurring the sender uses a *persist timer* that causes it to query the receiver periodically, to find out if the window has been increased. These segments from the sender are called *window probes*. In this chapter we'll examine window probes and the persist timer. We'll also examine the silly window syndrome, which is tied to the persist timer.

22.2 An Example

To see the persist timer in action we'll start a receiver process that listens for a connection request from a client, accepts the connection request, and then goes to sleep for a long time before reading from the network.

Our `sock` program lets us specify a pause option `-P` that sleeps between the server accepting the connection request and performing the first read. We'll invoke the server as:

```
bsdi % sock -i -s -P100000 5555
```

This has the server sleep for 100,000 seconds (27.8 hours) before reading from the network. The client is run in the host `svr4` and performs 1024-byte writes to port 5555 on the server. Figure 22.1 shows the `tcpdump` output. (We have removed the connection establishment from the output.)

```
 1    0.0                bsdi.1027 > svr4.5555: P 1:1025(1024) ack 1 win 4096
 2    0.191961 ( 0.1920) svr4.5555 > bsdi.1027: . ack 1025 win 4096
 3    0.196950 ( 0.0050) bsdi.1027 > svr4.5555: . 1025:2049(1024) ack 1 win 4096
 4    0.200340 ( 0.0034) bsdi.1027 > svr4.5555: . 2049:3073(1024) ack 1 win 4096
 5    0.207506 ( 0.0072) svr4.5555 > bsdi.1027: . ack 3073 win 4096
 6    0.212676 ( 0.0052) bsdi.1027 > svr4.5555: . 3073:4097(1024) ack 1 win 4096
 7    0.216113 ( 0.0034) bsdi.1027 > svr4.5555: P 4097:5121(1024) ack 1 win 4096
 8    0.219997 ( 0.0039) bsdi.1027 > svr4.5555: P 5121:6145(1024) ack 1 win 4096
 9    0.227882 ( 0.0079) svr4.5555 > bsdi.1027: . ack 5121 win 4096
10    0.233012 ( 0.0051) bsdi.1027 > svr4.5555: P 6145:7169(1024) ack 1 win 4096
11    0.237014 ( 0.0040) bsdi.1027 > svr4.5555: P 7169:8193(1024) ack 1 win 4096
12    0.240961 ( 0.0039) bsdi.1027 > svr4.5555: P 8193:9217(1024) ack 1 win 4096
13    0.402143 ( 0.1612) svr4.5555 > bsdi.1027: . ack 9217 win 0

14    5.351561 ( 4.9494) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
15    5.355571 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

16   10.351714 ( 4.9961) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
17   10.355670 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

18   16.351881 ( 5.9962) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
19   16.355849 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

20   28.352213 (11.9964) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
21   28.356178 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

22   52.352874 (23.9967) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
23   52.356839 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

24  100.354224 (47.9974) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
25  100.358207 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

26  160.355914 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
27  160.359835 ( 0.0039) svr4.5555 > bsdi.1027: . ack 9217 win 0

28  220.357575 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
29  220.361668 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0

30  280.359254 (59.9976) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
31  280.363315 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0
```

Figure 22.1 Example of persist timer probing a zero-sized window.

Segments 1–13 shows the normal data transfer from the client to the server, filling up the window with 9216 bytes of data. The server advertises a window of 4096, and has a default socket buffer size of 4096, but really accepts a total of 9216 bytes. This is some form of interaction between the TCP/IP code and the streams subsystem in SVR4.

In segment 13 the server acknowledges the previous four data segments, but advertises a window of 0, stopping the client from transmitting any more data. This causes the client to set its persist timer. If the client doesn't receive a window update when the timer expires, it probes the empty window, to see if a window update has been lost. Since our server process is asleep, the 9216 bytes of data are buffered by TCP, waiting for the application to issue a read.

Notice the spacing of the window probes by the client. The first (segment 14) is 4.949 seconds after receiving the zero-sized window. The next (segment 16) is 4.996 seconds later. The spacing is then about 6, 12, 24, 48, and 60 seconds after the previous.

Why are the spacings always a fraction of a second less than 5, 6, 12, 24, 48, and 60? These probes are triggered by TCP's 500-ms timer expiring. When the timer expires, the window probe is sent, and a reply is received about 4 ms later. The receipt of the reply causes the timer to be restarted, but the time until the next clock tick is about 500 minus 4 ms.

The normal TCP exponential backoff is used when calculating the persist timer. The first timeout is calculated as 1.5 seconds for a typical LAN connection. This is multiplied by 2 for a second timeout value of 3 seconds. A multiplier of 4 gives the next value of 6, a multiplier of 8 gives a value of 12, and so on. But the persist timer is always bounded between 5 and 60 seconds, which accounts for the values we see in Figure 22.1.

The window probes contain 1 byte of data (sequence number 9217). TCP is always allowed to send 1 byte of data beyond the end of a closed window. Notice, however, that the acknowledgments returned with the window size of 0 do not ACK this byte. (They ACK the receipt of all bytes through and including byte number 9216.) Therefore this byte keeps being retransmitted.

The characteristic of the persist state that is different from the retransmission timeout in Chapter 21 is that TCP *never* gives up sending window probes. These window probes continue to be sent at 60-second intervals until the window opens up or either of the applications using the connection is terminated.

22.3 Silly Window Syndrome

Window-based flow control schemes, such as the one used by TCP, can fall victim to a condition known as the *silly window syndrome* (SWS). When it occurs, small amounts of data are exchanged across the connection, instead of full-sized segments [Clark 1982].

It can be caused by either end: the receiver can advertise small windows (instead of waiting until a larger window could be advertised) and the sender can transmit small amounts of data (instead of waiting for additional data, to send a larger segment). Correct avoidance of the silly window syndrome is performed on both ends.

1. The receiver must not advertise small segments. The normal algorithm is for the receiver not to advertise a larger window than it is currently advertising (which can be 0) until the window can be increased by either one full-sized segment (i.e., the MSS being received) or by one-half the receiver's buffer space, whichever is smaller.

2. Sender avoidance of the silly window syndrome is done by not transmitting unless one of the following conditions is true: (a) a full-sized segment can be sent, (b) we can send at least one-half of the maximum sized window that the other end has ever advertised, or (c) we can send everything we have and either we are not expecting an ACK (i.e., we have no outstanding unacknowledged data) or the Nagle algorithm is disabled for this connection (Section 19.4).

   Condition (b) deals with hosts that always advertise tiny windows, perhaps smaller than the segment size. Condition (c) prevents us from sending small segments when we have unacknowledged data that is waiting to be ACKed and the Nagle algorithm is enabled. If the application is doing small writes (e.g., smaller than the segment size), it is condition (c) that avoids the silly window syndrome.

   These three conditions also let us answer the question: if the Nagle algorithm prevents us from sending small segments while there is outstanding unacknowledged data, how small is small? From condition (a) we see that "small" means the number of bytes is less than the segment size. Condition (b) only comes into play with older, primitive hosts.

Condition (b) in step 2 requires that the sender keep track of the maximum window size advertised by the other end. This is an attempt by the sender to guess the size of the other end's receive buffer. Although the size of the receiver buffer could decrease while the connection is established, in practice this is rare.

An Example

We'll now go through a detailed example to see the silly window syndrome avoidance in action, which also involves the persist timer. We'll use our `sock` program with the sending host, `sun`, doing six 1024-byte writes to the network:

```
sun % sock -i -n6 bsdi 7777
```

But we'll put some pauses in the receiving process on the host `bsdi`, pausing 4 seconds before doing the first read, and then pausing 2 seconds between successive reads. Additionally, the receiver issues 256-byte reads:

```
bsdi % sock -i -s -P4 -p2 -r256 7777
```

The reason for the initial pause is to let the receiver's buffer fill, forcing it to stop the transmitter. Since the receiver then performs small reads from the network, we expect to see the receiver perform silly window syndrome avoidance.

Figure 22.2 is the time line for the transfer of the 6144 bytes of data. (We have deleted the connection establishment.)

We also need to track what happens with the application that's reading the data at each point in time, along with the number of bytes currently in the receive buffer, and the number of bytes of available space in the receive buffer. Figure 22.3 shows everything that's happening.

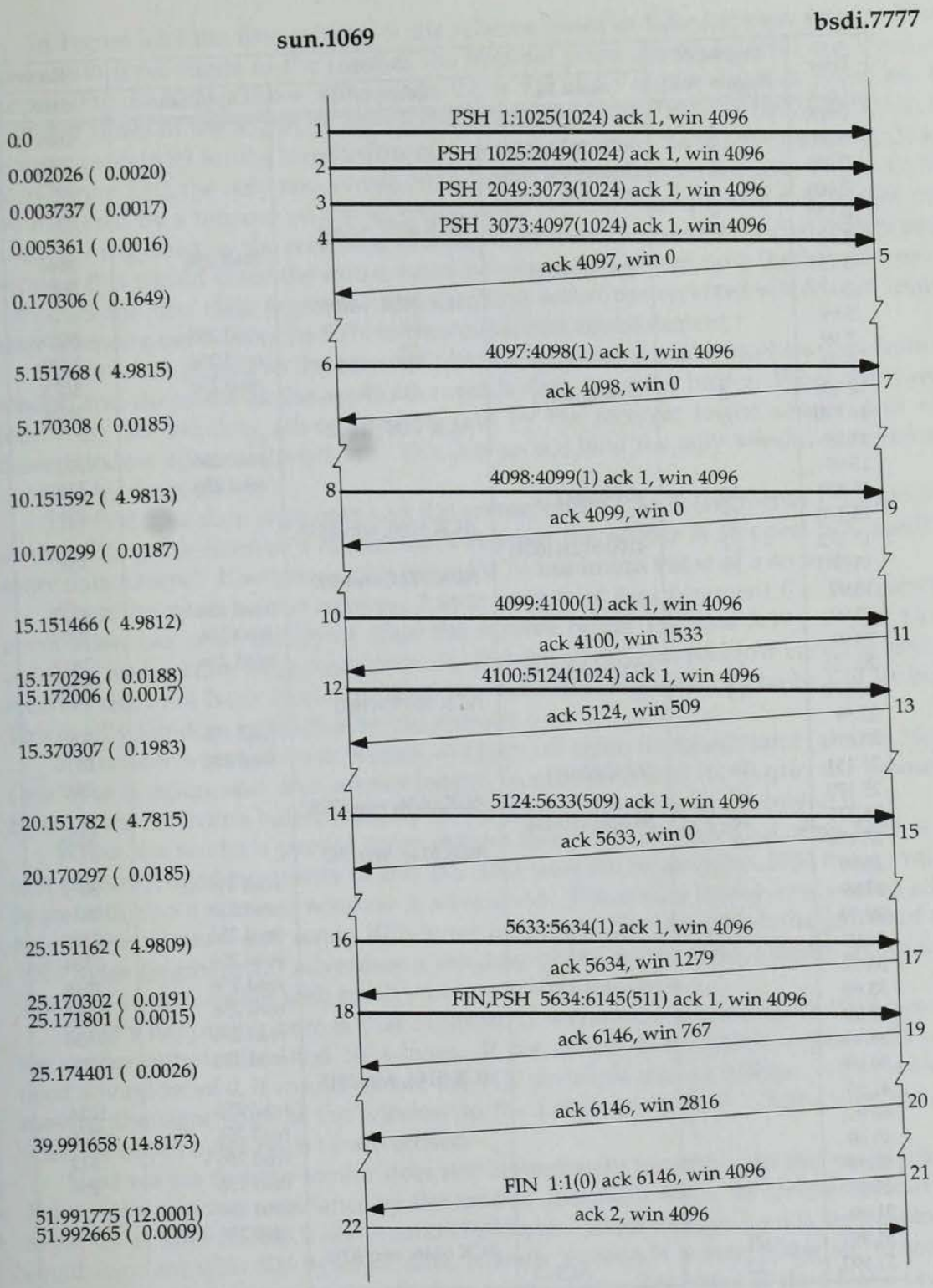


Figure 22.2 Time line showing receiver avoidance of silly window syndrome.

Time	Segment# (Figure 22.2)	Action			Receiver buffer	
		Send TCP	Receive TCP	Application	data	available
0.000	1	1:1025(1024)			1024	3072
0.002	2	1025:2049(1024)			2048	2048
0.003	3	2049:3073(1024)			3072	1024
0.005	4	3073:4097(1024)			4096	0
0.170	5		ACK 4097, win 0			
3.99				read 256	3840	256
5.151	6	4097:4098(1)			3841	255
5.17	7		ACK 4098, win 0			
5.99				read 256	3585	511
7.99				read 256	3329	767
9.99				read 256	3073	1023
10.151	8	4098:4099(1)			3074	1022
10.170	9		ACK 4099, win 0			
11.99				read 256	2818	1278
13.99				read 256	2562	1534
15.151	10	4099:4100(1)			2563	1533
15.170	11		ACK 4100, win 1533			
15.172	12	4100:5124(1024)			3587	509
15.370	13		ACK 5124, win 509			
15.99				read 256	3331	765
17.99				read 256	3075	1021
19.99				read 256	2819	1277
20.151	14	5124:5633(509)			3328	768
20.170	15		ACK 5633, win 0			
21.99				read 256	3072	1024
23.99				read 256	2816	1280
25.151	16	5633:5634(1)			2817	1279
25.170	17		ACK 5634, win 1279			
25.171	18	5634:6145(511)			3328	768
25.174	19		ACK 6146, win 767			
25.99				read 256	3072	1024
27.99				read 256	2816	1280
29.99				read 256	2560	1536
31.99				read 256	2304	1792
33.99				read 256	2048	2048
35.99				read 256	1792	2304
37.99				read 256	1536	2560
39.99				read 256	1280	2816
39.99	20		ACK 6146, win 2816			
41.99				read 256	1024	3072
43.99				read 256	768	3328
45.99				read 256	512	3584
47.99				read 256	256	3840
49.99				read 256	0	4096
51.99				read 256 (EOF)	0	4096
51.991	21		ACK 6146, win 4096			
51.992	22	ACK 2				

Figure 22.3 Sequence of events for receiver avoidance of the silly window syndrome.

In Figure 22.3 the first column is the relative point in time for each action. Those times with three digits to the right of the decimal point are taken from the `tcpdump` output (Figure 22.2). Those times with 99 to the right of the decimal point are the assumed times of the action on the receiving host. (Having these relative times on the receiver contain 99 for the hundredths of a second correlates them with segments 20 and 22 in Figure 22.2, the only two events on the receiver that we can see with `tcpdump` that are triggered by a timeout on the receiving host. All the other packets that we see from `bsdi` are triggered by the reception of a segment from the sender. It also makes sense, because this would place the initial 4-second pause just before time 0 when the sender transmits the first data segment. This is about when the receiver would get control, after receiving the ACK of its SYN in the connection establishment.)

The amount of data in the receiver's buffer increases when it receives data from the sender, and decreases as the application reads data from the buffer. What we want to follow are the window advertisements sent by the receiver to the sender, and what those window advertisements are. This lets us see how the silly window syndrome is avoided by the receiver.

The first four data segments and the corresponding ACK (segments 1–5) show the sender filling the receiver's buffer. At that point the sender is stopped but it still has more data to send. It sets its persist timer for its minimum value of 5 seconds.

When the persist timer expires, 1 byte of data is sent (segment 6). The receiving application has read 256 bytes from the receive buffer (at time 3.99), so the byte is accepted and acknowledged (segment 7). But the advertised window is still 0, since the receiver does not have room for either one full-sized segment or one-half of its buffer. This is silly window avoidance by the receiver.

The sender's persist timer is reset and goes off again 5 seconds later (at time 10.151). One byte is again sent and acknowledged (segments 8 and 9). Again the amount of room in the receiver's buffer (1022 bytes) forces it to advertise a window of 0.

When the sender's persist timer expires next, at time 15.151, another byte is sent and acknowledged (segments 10 and 11). This time the receiver has 1533 bytes available in its buffer, so a nonzero window is advertised. The sender immediately takes advantage of the window and sends 1024 bytes (segment 12). The acknowledgment of these 1024 bytes (segment 13) advertises a window of 509 bytes. This appears to contradict what we've seen earlier with small window advertisements.

What's happening here is that segment 11 advertised a window of 1533 bytes but the sender only transmitted 1024 bytes. If the acknowledgment in segment 13 advertised a window of 0, it would violate the TCP principle that a window cannot shrink by moving the right edge of the window to the left (Section 20.3). That's why the small window of 509 bytes must be advertised.

Next we see that the sender does not immediately transmit into this small window. This is silly window avoidance by the sender. Instead it waits for another persist timer to expire at time 20.151, when it sends 509 bytes. Even though it ends up sending this small segment with 509 bytes of data, it waits 5 seconds before doing so, to see if an ACK arrives that opens up the window more. These 509 bytes of data leave only 768 bytes of available space in the receive buffer, so the acknowledgment (segment 15) advertises a window of 0.

The persist timer goes off again at time 25.151, and the sender transmits 1 byte. The receive buffer then has 1279 bytes of space, which is the window advertised in segment 17.

The sender has only 511 additional bytes of data to transmit, which it sends immediately upon receiving the window advertisement of 1279 (segment 18). This segment also contains the FIN flag. The receiver acknowledges the data and the FIN, advertising a window of 767. (See Exercise 22.2.)

Since the sending application issues a close after performing its six 1024-byte writes, the sender's end of the connection goes from the ESTABLISHED state to the FIN_WAIT_1 state, to the FIN_WAIT_2 state (Figure 18.12). It sits in this state until receiving a FIN from the other end. There is no timer in this state (recall our discussion at the end of Section 18.6), since the FIN that it sent in segment 18 was acknowledged in segment 19. This is why we see no further transmissions by the sender until it receives the FIN (segment 21).

The receiving application continues reading 256 bytes of data every 2 seconds from the receive buffer. Why is the ACK sent at time 39.99 (segment 20)? The amount of room in the receive buffer has gone from its last advertised value of 767 (segment 19) to 2816 when the application reads at time 39.99. This equals 2049 bytes of additional space in the receive buffer. Recalling the first rule at the start of this section, the receiver now sends a window update because the amount of room has increased by one-half the room in the receive buffer. This implies that the receiving TCP checks whether to send a window update every time the application reads data from TCP's receive buffer.

The final application read occurs at time 51.99 and the application receives an end-of-file notification, since the buffer is empty. This causes the final two segments (21 and 22), which complete the termination of the connection.

22.4 Summary

TCP's persist timer is set by one end of a connection when it has data to send, but has been stopped because the other end has advertised a zero-sized window. The sender keeps probing the closed window using a retransmission interval similar to what we saw in Chapter 21. This probing of the closed window continues indefinitely.

When we ran an example to see the persist timer we also encountered TCP's avoidance of the silly window syndrome. This is to prevent TCP from advertising small windows or from sending small segments. In our example we saw avoidance of the silly window syndrome by both the sender and the receiver.

Exercises

22.1 In Figure 22.3 notice the times of all the acknowledgments (segments 5, 7, 9, 11, 13, 15, and 17): 0.170, 5.170, 10.170, 15.170, 15.370, 20.170, and 25.170. Also notice the time differences between receiving the data and sending the ACK: 164.5, 18.5, 18.7, 18.8, 198.3, 18.5, and 19.1 ms. Explain what's probably going on.

22.2 In Figure 22.3 at time 25.174 a window of 767 is advertised, but 768 bytes are available in the receive buffer. Why the difference of 1 byte?

23

TCP Keepalive Timer

23.1 Introduction

Many newcomers to TCP/IP are surprised to learn that no data whatsoever flows across an idle TCP connection. That is, if neither process at the ends of a TCP connection is sending data to the other, nothing is exchanged between the two TCP modules. There is no polling, for example, as you might find with other networking protocols. This means we can start a client process that establishes a TCP connection with a server, and walk away for hours, days, weeks or months, and the connection remains up. Intermediate routers can crash and reboot, phone lines may go down and back up, but as long as neither host at the ends of the connection reboots, the connection remains established.

This assumes that neither application—the client or server—has application-level timers to detect inactivity, causing either application to terminate. Recall at the end of Section 10.7 that BGP sends an application probe to the other end every 30 seconds. This is an application timer that is independent of the TCP keepalive timer.

There are times, however, when a server wants to know if the client's host has either crashed and is down, or crashed and rebooted. The *keepalive timer*, a feature of many implementations, provides this capability.

> Keepalives are not part of the TCP specification. The Host Requirements RFC provides three reasons not to use them: (1) they can cause perfectly good connections to be dropped during transient failures, (2) they consume unnecessary bandwidth, and (3) they cost money on an internet that charges by the packet. Nevertheless, many implementations provide the keepalive timer.

The keepalive timer is a controversial feature. Many feel that this polling of the other end has no place in TCP and should be done by the application, if desired. This is one of the *religious issues*, because of the fervor expressed by some on the topic.

The keepalive option can cause an otherwise good connection between two processes to be terminated because of a temporary loss of connectivity in the network joining the two end systems. For example, if the keepalive probes are sent during the time that an intermediate router has crashed and is rebooting, TCP will think that the client's host has crashed, which is not what has happened.

The keepalive feature is intended for server applications that might tie up resources on behalf of a client, and want to know if the client host crashes. Many versions of the Telnet server and Rlogin server enable the keepalive option by default.

A common example showing the need for the keepalive feature nowadays is when personal computer users use TCP/IP to login to a host using Telnet. If they just power off the computer at the end of the day, without logging off, they leave a half-open connection. In Figure 18.16 we showed how sending data across a half-open connection caused a reset to be returned, but that was from the client end, where the client was sending the data. If the client disappears, leaving the half-open connection on the server's end, and the server is waiting for some data from the client, the server will wait forever. The keepalive feature is intended to detect these half-open connections from the server side.

23.2 Description

In this description we'll call the end that enables the keepalive option the server, and the other end the client. There is nothing to stop a client from setting this option, but normally it's set by servers. It can also be set by both ends of a connection, if it's important for each end to know if the other end disappears. (In Chapter 29 we'll see that when NFS uses TCP, both the client and server set this option. But in Chapter 26 with Rlogin and Telnet, only the servers set the option, not the clients.)

If there is no activity on a given connection for 2 hours, the server sends a probe segment to the client. (We'll see what the probe segment looks like in the examples that follow.) The client host must be in one of four states.

1. The client host is still up and running and reachable from the server. The client's TCP responds normally and the server knows that the other end is still up. The server's TCP will reset the keepalive timer for 2 hours in the future. If there is application traffic across the connection before the next 2-hour timer expires, the timer is reset for 2 hours in the future, following the exchange of data.
2. The client's host has crashed and is either down or in the process of rebooting. In either case, its TCP is not responding. The server will not receive a response to its probe and it times out after 75 seconds. The server sends a total of 10 of these probes, 75 seconds apart, and if it doesn't receive a response, the server considers the client's host as down and terminates the connection.
3. The client's host has crashed and rebooted. Here the server will receive a response to its keepalive probe, but the response will be a reset, causing the server to terminate the connection.

4. The client's host is up and running, but unreachable from the server. This is the same as scenario 2, because TCP can't distinguish between the two. All it can tell is that no replies are received to its probes.

The server does not have to worry about the client's host being shut down and then rebooted. (This refers to an operator shutdown, instead of the host crashing.) When the system is shut down by an operator, all application processes are terminated (i.e., the client process), which causes the client's TCP to send a FIN on the connection. Receiving the FIN would cause the server's TCP to report an end-of-file to the server process, allowing the server to detect this scenario.

In the first scenario the server application has no idea that the keepalive probes are taking place. Everything is handled at the TCP layer. It's transparent to the application until one of scenarios 2, 3, or 4 occurs. In these three scenarios an error is returned to the server application by its TCP. (Normally the server has issued a read from the network, waiting for data from the client. If the keepalive feature returns an error, it is returned to the server as the return value from the read.) In scenario 2 the error is something like "connection timed out," and in scenario 3 we expect "connection reset by peer." The fourth scenario may look like the connection timed out, or may cause another error to be returned, depending on whether an ICMP error related to the connection is received. We look at all four scenarios in the next section.

> A perpetual question by people discovering the keepalive option is whether the 2-hour idle time value can be changed. They normally want it much lower, on the order of minutes. As we show in Appendix E, the value can usually be changed, but in all the systems described in this appendix, the keepalive interval is a system-wide value, so changing it affects all users of the option.
>
> The Host Requirements RFC says that an implementation may provide the keepalive feature, but it must not be enabled unless the application specifically says so. Also, the keepalive interval must be configurable, but it must default to no less than 2 hours.

23.3 Keepalive Examples

We'll now go through scenarios 2, 3, and 4 from the previous section, to see the packets exchanged using the keepalive option.

Other End Crashes

Let's see what happens when the client host crashes and does not reboot. To simulate this we'll do the following steps:

- Establish a connection between a client (our `sock` program on the host `bsdi`) and the standard echo server on the host `svr4`. The client enables the keepalive option with the `-K` option.
- Verify that data can go across the connection.
- Watch the client's TCP send keepalive packets every 2 hours, and see them acknowledged by the server's TCP.

- Disconnect the Ethernet cable from the server, and leave it off until the example is complete. This makes the client think the server host has crashed.
- We expect the server to send 10 keepalive probes, 75 seconds apart, before declaring the connection dead.

Here is the interactive output on the client:

```
bsdi % sock -K svr4 echo                  -K for keepalive option
hello, world                              type this at beginning, to verify connection is up
hello, world                              and see this echoed
                                          disconnect Ethernet cable after 4 hours
read error: Connection timed out          this happens about 6 hours and 10 minutes after start
```

Figure 23.1 shows the `tcpdump` output. (We have removed the connection establishment and the window advertisements.)

```
1      0.0                  bsdi.1055 > svr4.echo: P 1:14(13) ack 1
2      0.006105 (   0.0061) svr4.echo > bsdi.1055: P 1:14(13) ack 14
3      0.093140 (   0.0870) bsdi.1055 > svr4.echo: . ack 14

4   7199.972793 (7199.8797) arp who-has svr4 tell bsdi
5   7199.974878 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
6   7199.975741 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
7   7199.979843 (   0.0041) svr4.echo > bsdi.1055: . ack 14

8  14400.134330 (7200.1545) arp who-has svr4 tell bsdi
9  14400.136452 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
10 14400.137391 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
11 14400.141408 (   0.0040) svr4.echo > bsdi.1055: . ack 14

12 21600.318309 (7200.1769) arp who-has svr4 tell bsdi
13 21675.320373 (  75.0021) arp who-has svr4 tell bsdi
14 21750.322407 (  75.0020) arp who-has svr4 tell bsdi
15 21825.324460 (  75.0021) arp who-has svr4 tell bsdi
16 21900.436749 (  75.1123) arp who-has svr4 tell bsdi
17 21975.438787 (  75.0020) arp who-has svr4 tell bsdi
18 22050.440842 (  75.0021) arp who-has svr4 tell bsdi
19 22125.432883 (  74.9920) arp who-has svr4 tell bsdi
20 22200.434697 (  75.0018) arp who-has svr4 tell bsdi
21 22275.436788 (  75.0021) arp who-has svr4 tell bsdi
```

Figure 23.1 Keepalive packets that determine that a host has crashed.

Lines 1, 2, and 3 send the line "hello, world" from the client to the server and back. The first keepalive probe occurs 2 hours (7200 seconds) later on line 4. The first thing we see is an ARP request and an ARP reply, before the TCP segment on line 6 can be sent. The keepalive probe on line 6 elicits a response from the other end (line 7). The same sequence of packets is exchanged 2 hours later in lines 8–11.

If we could see all the fields in the keepalive probes, lines 6 and 10, we would see that the sequence number field is one less than the next sequence number to be sent (i.e., 13 in this example, when it should be 14), but because there is no data in the segment, `tcpdump` does not print the sequence number field. (It only prints the sequence number for empty segments that contain the SYN, FIN, or RST flags.) It is the receipt of this

incorrect sequence number that forces the server's TCP to respond with an ACK to the keepalive probe. The response tells the client the next sequence number that the server is expecting (14).

> Some older implementations based on 4.2BSD do not respond to these keepalive probes unless the segment contains data. Some systems can be configured to send one garbage byte of data in the probe to elicit the response. The garbage byte causes no harm, because it's not the expected byte (it's a byte that the receiver has previously received and acknowledged), so it's thrown away by the receiver. Other systems send the 4.3BSD-style segment (no data) for the first half of the probe period, and if no response is received, switch to the 4.2BSD-style segment for the last half.

We then disconnect the cable and expect the next probe, 2 hours later, to fail. When this next probe takes place, notice that we never see the TCP segments on the cable, because the host is not responding to ARP requests. We can still see that the client sends 10 probes, spaced 75 seconds apart, before giving up. We can see from our interactive script that the error code returned to the client process by TCP gets translated into "Connection timed out," which is what happened.

Other End Crashes and Reboots

In this example we'll see what happens when the client crashes and reboots. The initial scenario is the same as before, but after we verify that the connection is up, we disconnect the server from the Ethernet, reboot it, and then reconnect it to the Ethernet. We expect the next keepalive probe to generate a reset from the server, because the server now knows nothing about this connection. Here is the interactive session:

```
bsdi % sock -K svr4 echo              -K to enable keepalive option
hi there                              type this to verify connection is up
hi there                              and this is echoed back from other end
                                      here server is rebooted while disconnected from Ethernet
read error: Connection reset by peer
```

Figure 23.2 shows the `tcpdump` output. (We have removed the connection establishment and the window advertisements.)

```
1     0.0                    bsdi.1057 > svr4.echo: P 1:10(9) ack 1
2     0.006406 (   0.0064)   svr4.echo > bsdi.1057: P 1:10(9) ack 10
3     0.176922 (   0.1705)   bsdi.1057 > svr4.echo: . ack 10

4  7200.067151 (7199.8902)   arp who-has svr4 tell bsdi
5  7200.069751 (   0.0026)   arp reply svr4 is-at 0:0:c0:c2:9b:26
6  7200.070468 (   0.0007)   bsdi.1057 > svr4.echo: . ack 10
7  7200.075050 (   0.0046)   svr4.echo > bsdi.1057: R 1135563275:1135563275(0)
```

Figure 23.2 Keepalive example when other host has crashed and rebooted.

We establish the connection and send 9 bytes of data from the client to the server (lines 1–3). Two hours later the first keepalive probe is sent by the client, and the response is a reset from the server. The client application prints the error "Connection reset by peer," which makes sense.

Other End Is Unreachable

In this example the client has not crashed, but is not reachable during the 10-minute period when the keepalive probes are sent. An intermediate router may have crashed, a phone line may be temporarily out of order, or something similar.

To simulate this example we'll establish a TCP connection from our host `slip` through our dialup SLIP link to the host `vangogh.cs.berkeley.edu`, and then take the link down. First, here is the interactive output:

```
slip % sock -K vangogh.cs.berkeley.edu echo
testing                                  we type this line
testing                                  and see it echoed
                                         sometime in here the dialup SLIP link is taken down
read error: No route to host
```

Figure 23.3 shows the `tcpdump` output that was collected on the router `bsdi`. (The connection establishment and window advertisements have been removed.)

```
1      0.0                  slip.1056 > vangogh.echo: P 1:9(8) ack 1
2      0.277669 (   0.2777) vangogh.echo > slip.1056: P 1:9(8) ack 9
3      0.424423 (   0.1468) slip.1056 > vangogh.echo: . ack 9

4   7200.818081 (7200.3937) slip.1056 > vangogh.echo: . ack 9
5   7201.243046 (   0.4250) vangogh.echo > slip.1056: . ack 9

6  14400.688106 (7199.4451) slip.1056 > vangogh.echo: . ack 9
7  14400.689261 (   0.0012) sun > slip: icmp: net vangogh unreachable

8  14475.684360 (  74.9951) slip.1056 > vangogh.echo: . ack 9
9  14475.685504 (   0.0011) sun > slip: icmp: net vangogh unreachable

                            14 lines deleted

24 15075.759603 (  75.1008) slip.1056 > vangogh.echo: R 9:9(0) ack 9
25 15075.760761 (   0.0012) sun > slip: icmp: net vangogh unreachable
```

Figure 23.3 Keepalive example when other end is unreachable.

We start the example the same as before: lines 1–3 verify that the connection is up. The first keepalive probe 2 hours later is fine (lines 4 and 5), but before the next one occurs in another 2 hours, we bring down the SLIP connection between the routers `sun` and `netb`. (Refer to the inside front cover for the topology.)

The keepalive probe in line 6 elicits an ICMP network unreachable from the router `sun`. As we described in Section 21.10, this is just a soft error to the receiving TCP on the host `slip`. It records that the ICMP error was received, but the receipt of the error does not take down the connection. Nine more keepalive probes are sent, 75 seconds apart, before the sending host gives up. The error returned to the application generates a different message this time: "No route to host." We saw in Figure 6.12 (p. 82) that this corresponds to the ICMP network unreachable error.

23.4 Summary

As we said earlier, the keepalive feature is controversial. Protocol experts continue to debate whether it belongs in the transport layer, or should be handled entirely by the application.

It operates by sending a probe packet across a connection after the connection has been idle for 2 hours. Four different scenarios can occur: the other end is still there, the other end has crashed, the other end has crashed and rebooted, or the other end is currently unreachable. We saw each of these scenarios with an example, and saw different errors returned for the last three conditions.

In the first two examples that we looked at, had this feature not been provided, and without any application-level timer, our client would never have known that the other end had crashed, or crashed and rebooted. In the final example, however, nothing was wrong with the other end, the connection between them was temporarily down. We must be aware of this limitation when using keepalives.

Exercises

23.1 List some advantages of the keepalive feature.

23.2 List some disadvantages of the keepalive feature.

24

TCP Futures and Performance

24.1 Introduction

TCP has operated for many years over data links ranging from 1200 bits/sec dialup SLIP links to Ethernets. Ethernets were the predominant form of data link for TCP/IP in the 1980s and early 1990s. Although TCP operates correctly at speeds higher than an Ethernet (T3 phone lines, FDDI, and gigabit networks, for example), certain TCP limits start to be encountered at these higher speeds.

This chapter looks at some proposed modifications to TCP that allow it to obtain the maximum throughput at these higher speeds. We first look at the path MTU discovery mechanism, which we've seen earlier in the text, focusing this time on how it operates with TCP. This often lets TCP use an MTU greater than 536 for nonlocal connections, increasing its throughput.

We then look at long fat pipes, networks that have a large bandwidth-delay product, and the TCP limits that are encountered on these networks. Two new TCP options are described that deal with long fat pipes: a window scale option (to increase TCP's maximum window above 65535 bytes) and a timestamp option. This latter option lets TCP perform more accurate RTT measurement for data segments, and also provides protection against wrapped sequence numbers, which can occur at high speeds. These two options are defined in RFC 1323 [Jacobson, Braden, and Borman 1992].

We also look at the proposed T/TCP, modifications to TCP for transactions. The transaction mode of communication features a client request responded to by a server reply. It is a common paradigm for client–server computing. The goal of T/TCP is to reduce the number of segments exchanged by the two ends, avoiding the three-way handshake and the four segments to close the connection, so that the client receives the server's reply in one RTT plus the time required to process the request.

What is impressive about these new options—path MTU discovery, the window scale option, the timestamp option, and T/TCP—is that they are backward compatible with existing TCP implementations. Newer systems that include these options can still interoperate with all older systems. With the exception of an additional field in an ICMP message that can be used by path MTU discovery, these newer options need only be implemented on the end systems that want to take advantage of them.

We finish the chapter by looking at recently published figures dealing with TCP performance.

24.2 Path MTU Discovery

In Section 2.9 we described the concept of the *path MTU*. It is the minimum MTU on any network that is currently in the path between two hosts. Path MTU discovery entails setting the "don't fragment" (DF) bit in the IP header to discover if any router on the current path needs to fragment IP datagrams that we send. In Section 11.6 we showed the ICMP unreachable error returned by a router that is asked to forward an IP datagram with the DF bit set when the MTU is less than the datagram size. In Section 11.7 we showed a version of the `traceroute` program that used this mechanism to determine the path MTU to a destination. In Section 11.8 we saw how UDP handled path MTU discovery. In this section we'll examine how this mechanism is used by TCP, as specified by RFC 1191 [Mogul and Deering 1990].

> Of the various systems used in this text (see the Preface) only Solaris 2.x supports path MTU discovery.

TCP's path MTU discovery operates as follows. When a connection is established, TCP uses the minimum of the MTU of the outgoing interface, or the MSS announced by the other end, as the starting segment size. Path MTU discovery does not allow TCP to exceed the MSS announced by the other end. If the other end does not specify an MSS, it defaults to 536. It is also possible for an implementation to save path MTU information on a per-route basis, as we mentioned in Section 21.9.

Once the initial segment size is chosen, all IP datagrams sent by TCP on that connection have the DF bit set. If an intermediate router needs to fragment a datagram that has the DF bit set, it discards the datagram and generates the ICMP "can't fragment" error we described in Section 11.6.

If this ICMP error is received, TCP decreases the segment size and retransmits. If the router generated the newer form of this ICMP error, the segment size can be set to the next-hop MTU minus the sizes of the IP and TCP headers. If the older ICMP error is returned, the probable value of the next smallest MTU (Figure 2.5) must be tried. When a retransmission caused by this ICMP error occurs, the congestion window should not change, but slow start should be initiated.

Since routes can change dynamically, when some time has passed since the last decrease of the path MTU, a larger value (up to the minimum of the MSS announced by the other end, or the outgoing interface MTU) can be tried. RFC 1191 recommends this time interval be about 10 minutes. (We saw in Section 11.8 that Solaris 2.2 uses a 30-second timer for this.)

Given the normal default MSS of 536 for nonlocal destinations, path MTU discovery avoids fragmentation across intermediate links with an MTU of less than 576 (which is rare). It can also avoid fragmentation on local destinations when an intermediate link (e.g., an Ethernet) has a smaller MTU than the end-point networks (e.g., a token ring). But for path MTU discovery to be more useful, and take advantage of wide area networks with MTUs greater than 576, implementations must stop using a default MSS of 536 bytes for nonlocal destinations. A better choice for the MSS is the MTU of the outgoing interface (minus the size of the IP and TCP headers, of course). (In Appendix E we'll see that most implementations allow the system administrator to change this default MSS value.)

An Example

We can see how path MTU discovery operates when an intermediate router has an MTU less than either of the end point's interface MTUs. Figure 24.1 shows the topology for this example.

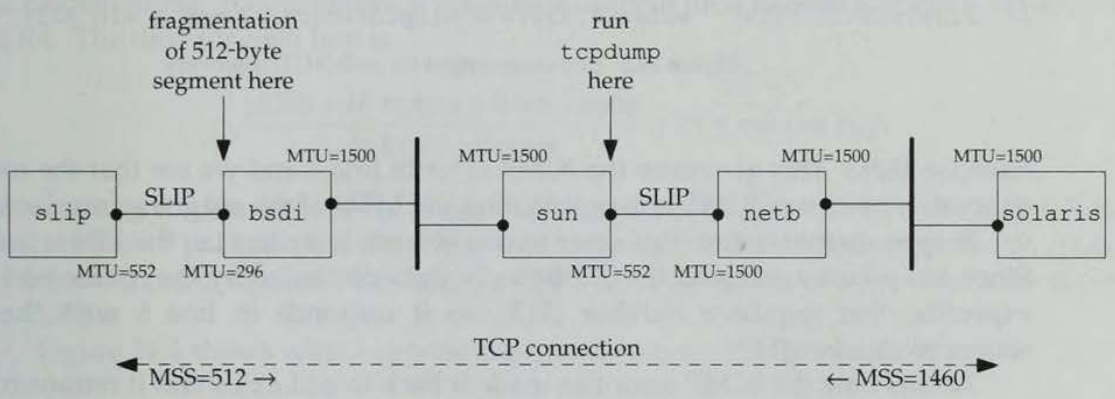


Figure 24.1 Topology for path MTU example.

We'll establish a connection from the host `solaris` (which supports the path MTU discovery mechanism) to the host `slip`. This setup is identical to the one used for our UDP path MTU discovery example (Figure 11.13) but here we have set the MTU of the interface on `slip` to 552, instead of its normal 296. This causes `slip` to announce an MSS of 512. But leaving the MTU of the SLIP link on `bsdi` at 296 will cause TCP segments greater than 256 to be fragmented, and we can see how the path MTU discovery mechanism on `solaris` handles this.

We'll run our `sock` program on `solaris` and perform one 512-byte write to the discard server on `slip`:

```
solaris % sock -i -n1 -w512 slip discard
```

Figure 24.2 shows the `tcpdump` output, collected on the SLIP interface on the host `sun`.

The MSS values in lines 1 and 2 are what we expect. We then see `solaris` send a 512-byte segment (line 3) containing the 512 bytes of data and the ACK of the SYN. (We saw this combination of the ACK of a SYN along with the first segment of data in

```
 1  0.0                  solaris.33016 > slip.discard: S 1171660288:1171660288(0)
                                                       win 8760 <mss 1460> (DF)
 2  0.101597 (0.1016)    slip.discard > solaris.33016: S 137984001:137984001(0)
                                                       ack 1171660289 win 4096
                                                       <mss 512>
 3  0.630609 (0.5290)    solaris.33016 > slip.discard: P 1:513(512)
                                                       ack 1 win 9216 (DF)
 4  0.634433 (0.0038)    bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 296 (DF)
 5  0.660331 (0.0259)    solaris.33016 > slip.discard: F 513:513(0)
                                                       ack 1 win 9216 (DF)
 6  0.752664 (0.0923)    slip.discard > solaris.33016: . ack 1 win 4096
 7  1.110342 (0.3577)    solaris.33016 > slip.discard: P 1:257(256)
                                                       ack 1 win 9216 (DF)
 8  1.439330 (0.3290)    slip.discard > solaris.33016: . ack 257 win 3840
 9  1.770154 (0.3308)    solaris.33016 > slip.discard: FP 257:513(256)
                                                        ack 1 win 9216 (DF)
10  2.095987 (0.3258)    slip.discard > solaris.33016: . ack 514 win 3840
11  2.138193 (0.0422)    slip.discard > solaris.33016: F 1:1(0) ack 514 win 4096
12  2.310103 (0.1719)    solaris.33016 > slip.discard: . ack 2 win 9216 (DF)
```

Figure 24.2 `tcpdump` output for path MTU discovery.

Exercise 18.9.) This generates the ICMP error in line 4 and we see that the router `bsdi` generates the newer ICMP error containing the MTU of the outgoing interface.

It appears that before this error makes it back to `solaris`, the FIN is sent (line 5). Since `slip` never received the 512 bytes of data discarded by the router `bsdi`, it is not expecting this sequence number (513), so it responds in line 6 with the expected sequence number (1).

At this time the ICMP error has made it back to `solaris` and it retransmits the 512 bytes of data in two 256-byte segments (lines 7 and 9). Both are sent with the DF bit set, since there could be another router beyond `bsdi` with a smaller MTU.

A longer transfer was run (taking about 15 minutes) and after moving from the 512-byte initial segment to 256-byte segments, `solaris` never tried the higher segment size again.

Big Packets or Small Packets?

Conventional wisdom says that bigger packets are better [Mogul 1993, Sec. 15.2.8] because sending fewer big packets "costs less" than sending more smaller packets. (This assumes the packets are not large enough to cause fragmentation, since that introduces another set of problems.) The reduced cost is that associated with the network (packet header overhead), routers (routing decisions), and hosts (protocol processing and device interrupts). Not everyone agrees with this [Bellovin 1993].

Consider the following example. We send 8192 bytes through four routers, each connected with a T1 telephone line (1,544,000 bits/sec). First we use two 4096-byte packets, as shown in Figure 24.3.

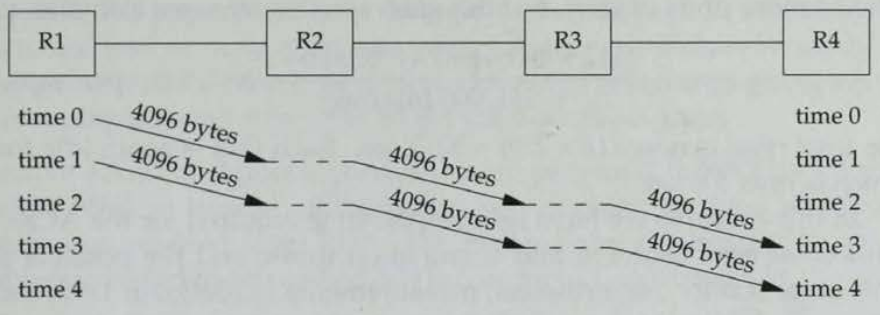


Figure 24.3 Sending two 4096-byte packets through four routers.

The basic problem is that routers are store-and-forward devices. They normally receive the entire input packet, validate the IP header including the IP checksum, make their routing decision, and start sending the output packet. In this figure we're assuming the ideal case where it takes no time for these operations to occur at the router (the horizontal dashed lines). Nevertheless, it takes four units of time to send all 8192 bytes from R1 to R4. The time for each hop is

$$\frac{(4096 + 40 \text{ bytes}) \times 8 \text{ bits/byte}}{1,544,000 \text{ bits/sec}} = 21.4 \text{ ms per hop}$$

(We account for the 40 bytes of IP and TCP header.) The total time to send the data is the number of packets plus the number of hops, minus one, which we can see visually in this example is four units of time, or 85.6 ms. Each link is idle for two units of time, or 42.8 ms.

Figure 24.4 shows what happens if we send sixteen 512-byte packets.

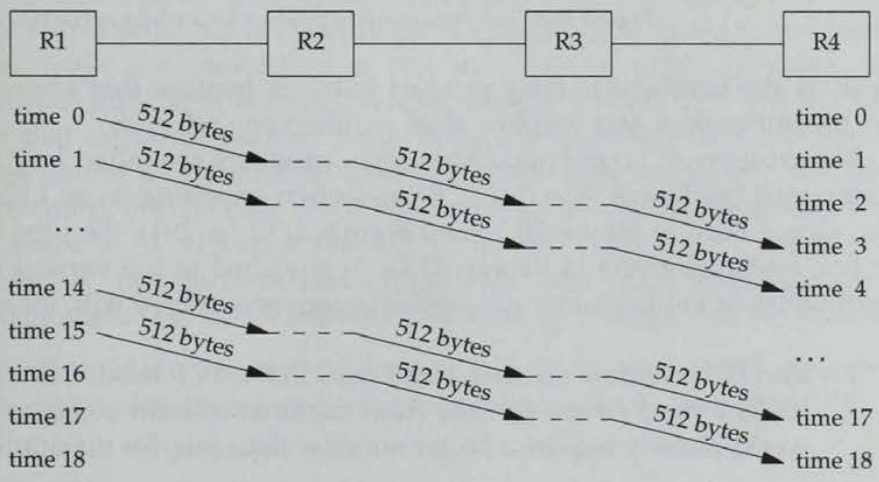


Figure 24.4 Sending sixteen 512-byte packets through four routers.

It takes more units of time, but the units are shorter since a smaller packet is being sent.

$$\frac{(512+40 \text{ bytes}) \times 8 \text{ bits/byte}}{1{,}544{,}000 \text{ bits/sec}} = 2.9 \text{ ms per hop}$$

The total time is now $(18 \times 2.9) = 52.2$ ms. Each link is again idle for two units of time, which is now 5.8 ms.

In this example we have ignored the time required for the ACKs to be returned, the connection establishment and termination times, and the possible sharing of the links with other traffic. Nevertheless, measurements in [Bellovin 1993] indicate that bigger is not always better. More research is required in this area on various networks.

24.3 Long Fat Pipes

In Section 20.7 we showed the capacity of a connection as

capacity (bits) = *bandwidth* (bits/sec) × *round-trip time* (sec)

and called this the *bandwidth-delay product*. This is also called the size of the pipe between the end points.

Existing limits in TCP are being encountered as this product increases to larger and larger values. Figure 24.5 shows some values for various types of networks.

Network	Bandwidth (bits/sec)	Round-trip time (ms)	Bandwidth-delay product (bytes)
Ethernet LAN	10,000,000	3	3,750
T1 telephone line, transcontinental	1,544,000	60	11,580
T1 telephone line, satellite	1,544,000	500	95,500
T3 telephone line, transcontinental	45,000,000	60	337,500
gigabit, transcontinental	1,000,000,000	60	7,500,000

Figure 24.5 Bandwidth-delay product for various networks.

We show the bandwidth-delay product in bytes, because that's how we typically measure the buffer sizes and window sizes required on each end.

Networks with large bandwidth-delay products are called *long fat networks* (LFNs, pronounced "elefan(t)s"), and a TCP connection operating on an LFN is called a *long fat pipe*. Going back to Figure 20.11 and Figure 20.12 (p. 291), the pipe can be stretched in the horizontal direction (a longer RTT), or stretched in the vertical direction (a higher bandwidth), or both. Numerous problems are encountered with long fat pipes.

1. The TCP window size is a 16-bit field in the TCP header, limiting the window to 65535 bytes. As we can see from the final column in Figure 24.5, existing networks already require a larger window than this, for maximum throughput.

   The window scale option described in Section 24.4 solves this problem.

2. Packet loss in an LFN can reduce throughput drastically. If only a single segment is lost, the fast retransmit and fast recovery algorithm that we described in

Section 21.7 is required to keep the pipe from draining. But even with this algorithm, the loss of more than one packet within a window typically causes the pipeline to drain. (If the pipe drains, slow start gets things going again, but that takes multiple round-trip times to get the pipe filled again.)

Selective acknowledgments (SACKs) were proposed in RFC 1072 [Jacobson and Braden 1988] to handle multiple dropped packets within a window. But this feature was omitted from RFC 1323, because the authors felt several technical problems needed to be worked out before including them in TCP.

3. We saw in Section 21.4 that many TCP implementations only measure one round-trip time per window. They do not measure the RTT of every segment. Better RTT measurements are required for operating on an LFN.

   The timestamp option, which we describe in Section 24.5, allows more segments to be timed, including retransmissions.

4. TCP identifies each byte of data with a 32-bit unsigned sequence number. What's to prevent a segment that gets delayed in the network from reappearing at a later time, after the connection that it was associated with has been terminated, and after a new connection has been established between the same two hosts and port numbers? First recall that the TTL field in the IP header puts an upper bound on the lifetime of any IP datagram—255 hops or 255 seconds, whichever comes first. In Section 18.6 we defined the maximum segment lifetime (MSL) as an implementation parameter used to prevent this scenario from happening. The recommended value of the MSL is 2 minutes (giving a 2MSL of 240 seconds), but we saw in Section 18.6 that many implementations use an MSL value of 30 seconds.

   A different problem with TCP's sequence numbers appears with LFNs. Since the sequence number space is finite, the same sequence number is reused after 4,294,967,296 bytes have been transmitted. What if a segment containing the byte with a sequence number N gets delayed in the network and then reappears later, while the connection is still up? This is only a problem if the same sequence number N is reused within the MSL period, that is, if the network is so fast that sequence number wrap occurs in less than MSL. On an Ethernet it takes almost 60 minutes to send this much data, so there is no chance of this happening, but the time required for the wrap to occur drops as the bandwidth increases: a T3 telephone line (45 Mbits/sec) wraps in 12 minutes, FDDI (100 Mbits/sec) in 5 minutes, and a gigabit network (1000 Mbits/sec) in 34 seconds. The problem here is not the bandwidth-delay product, but the bandwidth itself.

   In Section 24.6 we describe a way to handle this: the PAWS algorithm (protection against wrapped sequence numbers), which uses the TCP timestamp option.

4.4BSD contains all the options and algorithms that we describe in the following sections: the window scale option, the timestamp option, and the protection against wrapped sequence numbers. Numerous vendors are also starting to support these options.

Gigabit Networks

When networks reach gigabit speeds, things change. [Partridge 1994] covers gigabit networks in detail. Here we'll look at the differences between latency and bandwidth [Kleinrock 1992].

Consider sending a one million byte file across the United States, assuming a 30-ms latency. Figure 24.6 shows two scenarios, the top illustration uses a T1 telephone line (1,544,000 bits/sec) and the bottom uses a 1 gigabit/sec network. Time is shown along the x-axis, with the sender on the left and the receiver on the right, and capacity on the y-axis. The shaded area in both pictures is the one million bytes to send.

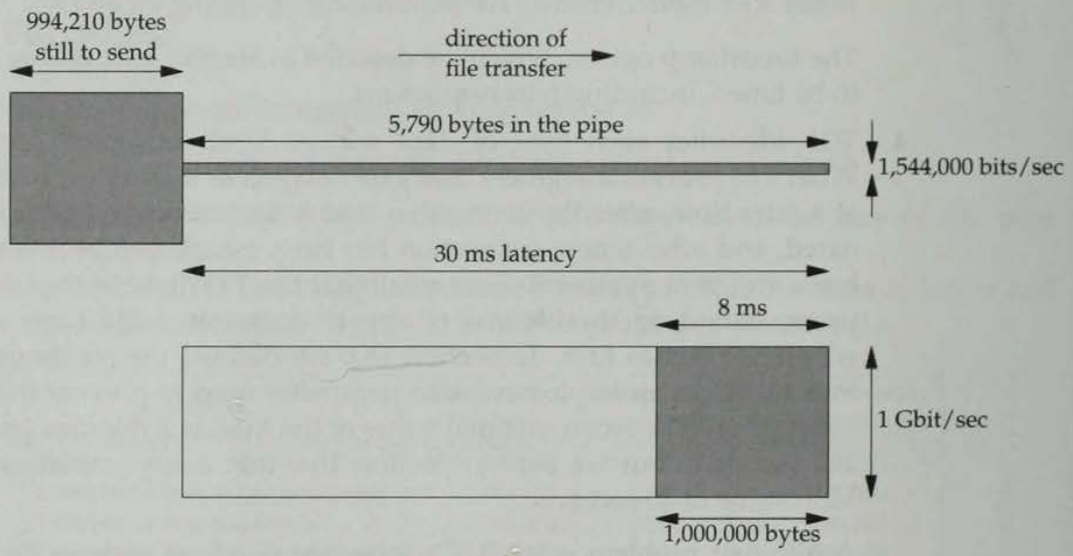


Figure 24.6 Sending a 1-Mbyte file across networks with a 30-ms latency.

Figure 24.6 shows the status of both networks after 30 ms. With both networks the first bit of data reaches the other end after 30 ms (the latency), but with the T1 network the capacity of the pipe is only 5,790 bytes, so 994,210 bytes are still at the sender, waiting to be sent. The capacity of the gigabit network, however, is 3,750,000 bytes, so the entire file uses just over 25% of the pipe. The last bit of the file reaches the receiver 8 ms after the first bit.

The total time to transfer the file across the T1 network is 5.211 seconds. If we throw more bandwidth at the problem, a T3 network (45,000,000 bits/sec), the total time decreases to 0.208 seconds. Increasing the bandwidth by a factor of 29 reduces the total time by a factor of 25.

With the gigabit network the total time to transfer the file is 0.038 seconds: the 30-ms latency plus the 8 ms for the actual file transfer. Assuming we could double the bandwidth to 2 gigabits/sec, we only reduce the total time to 0.034 seconds: the same 30-ms latency plus 4 ms to transfer the file. Doubling the bandwidth now decreases the total time by only 10%. At gigabit speeds we are latency limited, not bandwidth limited.

The latency is caused by the speed of light and can't be decreased (unless Einstein was wrong). The effect of this fixed latency becomes worse when we consider the packets required to establish and terminate a connection. Gigabit networks will cause several networking issues to be looked at differently.

24.4 Window Scale Option

The window scale option increases the definition of the TCP window from 16 to 32 bits. Instead of changing the TCP header to accommodate the larger window, the header still holds a 16-bit value, and an option is defined that applies a scaling operation to the 16-bit value. TCP then maintains the "real" window size internally as a 32-bit value.

We saw an example of this option in Figure 18.20 (p. 253). The 1-byte shift count is between 0 (no scaling performed) and 14. This maximum value of 14 is a window of 1,073,725,440 bytes (65535×2^{14}).

This option can only appear in a SYN segment; therefore the scale factor is fixed in each direction when the connection is established. To enable window scaling, both ends must send the option in their SYN segments. The end doing the active open sends the option in its SYN, but the end doing the passive open can send the option only if the received SYN specifies the option. The scale factor can be different in each direction.

If the end doing the active open sends a nonzero scale factor, but doesn't receive a window scale option from the other end, it sets its send and receive shift count to 0. This lets newer systems interoperate with older systems that don't understand the new option.

> The Host Requirements RFC requires TCP to accept an option in any segment. (The only previously defined option, the maximum segment size, only appeared in SYN segments.) It further requires TCP to ignore any option it doesn't understand. This is made easy since all the new options have a length field (Figure 18.20, p. 253).

Assume we are using the window scale option, with a shift count of S for sending and a shift count of R for receiving. Then every 16-bit advertised window that we receive from the other end is left shifted by R bits to obtain the real advertised window size. Every time we send a window advertisement to the other end, we take our real 32-bit window size and right shift it S bits, placing the resulting 16-bit value in the TCP header.

The shift count is automatically chosen by TCP, based on the size of the receive buffer. The size of this buffer is set by the system, but the capability is normally provided for the application to change it. (We discussed this buffer in Section 20.4.)

An Example

If we initiate a connection using our `sock` program from the 4.4BSD host `vangogh.cs.berkeley.edu`, we can see its TCP calculate the window scale factor. The following interactive output shows two consecutive runs of the program, first specifying a receive buffer of 128000 bytes, and then a receive buffer of 220000 bytes:

```
vangogh % sock -v -R128000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 128000
connected on 128.32.130.2.4107 to 140.252.13.35.7
TCP_MAXSEG = 512
hello, world                          we type this line
hello, world                          and it's echoed here
^D                                    type end-of-file character to terminate

vangogh % sock -v -R220000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 220000
connected on 128.32.130.2.4108 to 140.252.13.35.7
TCP_MAXSEG = 512
bye, bye                              type this line
bye, bye                              and it's echoed here
^D                                    type end-of-file character to terminate
```

Figure 24.7 shows the `tcpdump` output for these two connections. (We have deleted the final 8 lines for the second connection, because nothing new is shown.)

```
 1   0.0                  vangogh.4107 > bsdi.echo: S 462402561:462402561(0)
                            win 65535
                            <mss 512,nop,wscale 1,nop,nop,timestamp 995351 0>
 2   0.003078 ( 0.0031)   bsdi.echo > vangogh.4107: S 177032705:177032705(0)
                            ack 462402562 win 4096 <mss 512>
 3   0.300255 ( 0.2972)   vangogh.4107 > bsdi.echo: . ack 1 win 65535

 4  16.920087 (16.6198)   vangogh.4107 > bsdi.echo: P 1:14(13) ack 1 win 65535
 5  16.923063 ( 0.0030)   bsdi.echo > vangogh.4107: P 1:14(13) ack 14 win 4096
 6  17.220114 ( 0.2971)   vangogh.4107 > bsdi.echo: . ack 14 win 65535

 7  26.640335 ( 9.4202)   vangogh.4107 > bsdi.echo: F 14:14(0) ack 14 win 65535
 8  26.642688 ( 0.0024)   bsdi.echo > vangogh.4107: . ack 15 win 4096
 9  26.643964 ( 0.0013)   bsdi.echo > vangogh.4107: F 14:14(0) ack 15 win 4096
10  26.880274 ( 0.2363)   vangogh.4107 > bsdi.echo: . ack 15 win 65535

11  44.400239 (17.5200)   vangogh.4108 > bsdi.echo: S 468226561:468226561(0)
                            win 65535
                            <mss 512,nop,wscale 2,nop,nop,timestamp 995440 0>
12  44.403358 ( 0.0031)   bsdi.echo > vangogh.4108: S 182792705:182792705(0)
                            ack 468226562 win 4096 <mss 512>
13  44.700027 ( 0.2967)   vangogh.4108 > bsdi.echo: . ack 1 win 65535
                          remainder of this connection deleted
```

Figure 24.7 Example of window scale option.

In line 1 `vangogh` advertises a window of 65535 and specifies the window scale option with a shift count of 1. This advertised window is the largest possible value that is less than the receive buffer size (128000), because the window field in a SYN segment is never scaled.

The scale factor of 1 means `vangogh` would like to send window advertisements up to 131070 (65535×2^1). This will accommodate our receive buffer size (128000). Since `bsdi` does not send the window scale option in its SYN (line 2), the option is not used.

Notice that vangogh continues to use the largest window possible (65535) for the remainder of the connection.

For the second connection vangogh requests a shift count of 2, meaning it would like to send window advertisements up to 262140 (65535×2^2), which is greater than our receive buffer size (220000).

24.5 Timestamp Option

The timestamp option lets the sender place a timestamp value in every segment. The receiver reflects this value in the acknowledgment, allowing the sender to calculate an RTT for each received ACK. (We must say "each received ACK" and not "each segment" since TCP normally acknowledges multiple segments per ACK.) We said that many current implementations only measure one RTT per window, which is OK for windows containing eight segments. Larger window sizes, however, require better RTT calculations.

> Section 3.1 of RFC 1323 gives the signal processing reasons for requiring better RTT estimates for bigger windows. Basically the RTT is measured by sampling a data signal (the data segments) at a lower frequency (once per window). This introduces *aliasing* into the estimated RTT. When there are eight segments per window, the sample rate is one-eighth the data rate, which is tolerable, but with 100 segments per window, the sample rate is 1/100th the data rate. This can cause the estimated RTT to be inaccurate, resulting in unnecessary retransmissions. If a segment is lost, it only gets worse.

Figure 18.20 showed the format of the timestamp option. The sender places a 32-bit value in the first field, and the receiver echoes this back in the reply field. TCP headers containing this option will increase from the normal 20 bytes to 32 bytes.

The timestamp is a monotonically increasing value. Since the receiver echoes what it receives, the receiver doesn't care what the timestamp units are. This option does not require any form of clock synchronization between the two hosts. RFC 1323 recommends that the timestamp value increment by one between 1 ms and 1 second.

> 4.4BSD increments the timestamp clock once every 500 ms and this timestamp clock is reset to 0 on a reboot.
>
> In Figure 24.7, if we look at the timestamp in segment 1 and the timestamp in segment 11, the difference (89 units) corresponds to 500 ms per unit for the time difference of 44.4 seconds.

The specification of this option during connection establishment is handled the same way as the window scale option in the previous section. The end doing the active open specifies the option in its SYN. Only if it receives the option in the SYN from the other end can the option be sent in future segments.

We've seen that a receiving TCP does not have to acknowledge every data segment that it receives. Many implementations send an ACK for every other data segment. If the receiver sends an ACK that acknowledges two received data segments, which received timestamp is sent back in the timestamp echo reply field?

To minimize the amount of state maintained by either end, only a single timestamp value is kept per connection. The algorithm to choose when to update this value is simple.

1. TCP keeps track of the timestamp value to send in the next ACK (a variable named *tsrecent*) and the acknowledgment sequence number from the last ACK that was sent (a variable named *lastack*). This sequence number is the next sequence number the receiver is expecting.
2. When a segment arrives, if the segment contains the byte numbered *lastack*, then the timestamp value from the segment is saved in *tsrecent*.
3. Whenever a timestamp option is sent, *tsrecent* is sent as the timestamp echo reply field and the sequence number field is saved in *lastack*.

This algorithm handles the following two cases:

1. If ACKs are delayed by the receiver, the timestamp value returned as the echo value will correspond to the earliest segment being acknowledged.

   For example, if two segments containing bytes 1–1024 and 1025–2048 arrive, both with a timestamp option, and the receiver acknowledges them both with an ACK 2049, the timestamp in the ACK will be the value from the first segment containing bytes 1–1024. This is correct because the sender must calculate its retransmission timeout taking the delayed ACKs into consideration.
2. If a received segment is in-window but out-of-sequence, implying that a previous segment has been lost, when that missing segment is received, its timestamp will be echoed, not the timestamp from the out-of-sequence segment.

   For example, assume three segments, each containing 1024 bytes, are received in the following order: segment 1 with bytes 1–1024, segment 3 with bytes 2049–3072, then segment 2 with bytes 1025–2048. The ACKs sent back will be ACK 1025 with the timestamp from segment 1 (a normal ACK for data that was expected), ACK 1025 with the timestamp from segment 1 (a duplicate ACK in response to the in-window but the out-of-sequence segment), then ACK 3073 with the timestamp from segment 2 (not the later timestamp from segment 3). This has the effect of overestimating the RTT when segments are lost, which is better than underestimating it. Also, if the final ACK contained the timestamp from segment 3, it might include the time required for the duplicate ACK to be returned and segment 2 to be retransmitted, or it might include the time for the sender's retransmission timeout for segment 2 to expire. In either case, echoing the timestamp from segment 3 could bias the sender's RTT calculations.

Although the timestamp option allows for better RTT calculations, it also provides a way for the receiver to avoid receiving old segments and considering them part of the existing data segment. The next section describes this.

24.6 PAWS: Protection Against Wrapped Sequence Numbers

Consider a TCP connection using the window scale option with the largest possible window, 1 gigabyte (2^{30}). (The largest window is just smaller than this, 65535×2^{14}, not $2^{16} \times 2^{14}$, but that doesn't affect this discussion.) Also assume the timestamp option is being used and that the timestamp value assigned by the sender increments by one for each window that is sent. (This is conservative. Normally the timestamp increments faster than this.) Figure 24.8 shows the possible data flow between the two hosts, when transferring 6 gigabytes. To avoid lots of 10-digit numbers, we use the notation G to mean a multiple of 1,073,741,824. We also use the notation from `tcpdump` that *J:K* means byte *J* through and including byte *K–1*.

Time	Bytes sent	Send sequence#	Send timestamp	Receive
A	0G:1G	0G:1G	1	OK
B	1G:2G	1G:2G	2	OK but one segment lost and retransmitted
C	2G:3G	2G:3G	3	OK
D	3G:4G	3G:4G	4	OK
E	4G:5G	0G:1G	5	OK
F	5G:6G	1G:2G	6	OK but retransmitted segment reappears

Figure 24.8 Transferring 6 gigabytes in six 1-gigabyte windows.

The 32-bit sequence number wraps between times D and E. We assume that one segment gets lost at time B and is retransmitted. We also assume that this lost segment reappears at time F.

This assumes that the time difference between the segment getting lost and reappearing is less than the MSL; otherwise the segment would have been discarded by some router when its TTL expired. As we mentioned earlier, it is only with high-speed connections that this problem appears, where old segments can reappear and contain sequence numbers currently being transmitted.

We can also see from Figure 24.8 that using the timestamp prevents this problem. The receiver considers the timestamp as a 32-bit extension of the sequence number. Since the lost segment that reappears at time F has a timestamp of 2, which is less than the most recent valid timestamp (5 or 6), it is discarded by the PAWS algorithm.

The PAWS algorithm does not require any form of time synchronization between the sender and receiver. All the receiver needs is for the timestamp values to be monotonically increasing, and to increase by at least one per window.

24.7 T/TCP: A TCP Extension for Transactions

TCP provides a *virtual-circuit* transport service. There are three distinct phases in the life of a connection: establishment, data transfer, and termination. Applications such as remote login and file transfer are well suited to a virtual-circuit service.

Other applications, however, are designed to use a transaction service. A *transaction* is a client request followed by a server response with the following characteristics:

1. The overhead of connection establishment and connection termination should be avoided. When possible, send one request packet and receive one reply packet.
2. The latency should be reduced to RTT plus SPT, where RTT is the round-trip time and SPT is the server processing time to handle the request.
3. The server should detect duplicate requests and not replay the transaction when a duplicate request arrives. (Avoiding the replay means the server does not process the request again. The server sends back the saved reply corresponding to that request.)

One application that we've already seen that uses this type of service is the Domain Name System (Chapter 14), although the DNS is not concerned with the server replaying duplicate requests.

Today the choice an application designer has is TCP or UDP. TCP provides too many features for transactions, and UDP doesn't provide enough. Usually the application is built using UDP (to avoid the overhead of TCP connections) but many of the desirable features (dynamic timeout and retransmission, congestion avoidance, etc.) are placed into the application, where they're reinvented over and over again.

A better solution is to provide a transport layer that provides efficient handling of transactions. The transaction protocol we describe in this section is called T/TCP. Our description is from its definition, RFC 1379 [Braden 1992b] and [Braden 1992c].

Most TCPs require 7 segments to open and close a connection (see Figure 18.13, p. 242). Three more segments are then added: one with the request, another with the reply and an ACK of the request, and a third with the ACK of the reply. If additional control bits are added onto the segments—that is, the first segment contains a SYN, the client request, and a FIN—the client still sees a minimal overhead of twice the RTT plus SPT. (Sending a SYN along with data and a FIN is legal; whether current TCPs handle it correctly is another question.)

Another problem with TCP is the TIME_WAIT state and its required 2MSL wait. As shown in Exercise 18.14, this limits the transaction rate between two hosts to about 268 per second.

The two modifications required for TCP to handle transactions are to avoid the three-way handshake and shorten the TIME_WAIT state. T/TCP avoids the three-way handshake by using an accelerated open:

1. It assigns a 32-bit *connection count* (CC) value to connections it opens, either actively or passively. A host's CC value is assigned from a global counter that gets incremented by 1 each time it's used.
2. Every segment between two hosts using T/TCP includes a new TCP option named CC. This option has a length of 6 bytes and contains the sender's 32-bit CC value for the connection.

3. A host maintains a per-host cache of the last CC value received in an acceptable SYN segment from that host.
4. When a CC option is received on an initial SYN, the receiver compares the value with the cached value for the sender. If the received CC is greater than the cached CC, the SYN is new and any data in the segment is passed to the receiving application (the server). The connection is called *half-synchronized*.

   If the received CC is not greater than the cached CC, or if the receiving host doesn't have a cached CC for this client, the normal TCP three-way handshake is performed.
5. The SYN, ACK segment in response to an initial SYN echoes the received CC value in another new option named CCECHO.
6. The CC value in a non-SYN segment detects and rejects any duplicate segments from previous incarnations of the same connection.

The accelerated open avoids the need for a three-way handshake unless either the client or server has crashed and rebooted. The cost is that the server must remember the last CC received from each client.

The TIME_WAIT state is shortened by calculating the TIME_WAIT delay dynamically, based on the measured RTT between the two hosts. The TIME_WAIT delay is set to 8 times *RTO*, the retransmission timeout value (Section 21.3).

Using these features the minimal transaction sequence is an exchange of three segments:

1. Client to server, caused by an active open: client-SYN, client-data (the request), client-FIN, and client-CC.

   When the server TCP with the passive open receives this segment, if the client-CC is greater than the cached CC for this client host, the client-data is passed to the server application, which processes the request.
2. Server to client: server-SYN, server-data (reply), server-FIN, ACK of client-FIN, server-CC, and CCECHO of client-CC. Since TCP acknowledgments are cumulative, this ACK of the client FIN acknowledges the client's SYN, data, and FIN.

   When the client TCP receives this segment it passes the reply to the client application.
3. Client to server: ACK of server-FIN, which acknowledges the server's SYN, data, and FIN.

The client's response time to its request is RTT plus SPT.

There are many fine points to the implementation of this TCP option that are covered in the references. We summarize them here:

- The server's SYN, ACK (the second segment) should be delayed, to allow the reply to piggyback with it. (Normally the ACK of a SYN is not delayed.) It can't delay too long, or the client will time out and retransmit.

- The request can require multiple segments, but the server must handle their possible out-of-order arrival. (Normally when data arrives before the SYN, the data is discarded and a reset is generated. With T/TCP this out-of-order data should be queued instead.)
- The API must allow the server process to send data and close the connection in a single operation to allow the FIN in the second segment to piggyback with the reply. (Normally the application would write the reply, causing a data segment to be sent, and then close the connection, causing the FIN to be sent.)
- The client is sending data in the first segment before receiving an MSS announcement from the server. To avoid restricting the client to an MSS of 536, the MSS for a given host should be cached along with its CC value.
- The client is also sending data to the server without receiving a window advertisement from the server. T/TCP suggests a default window of 4096 bytes and also caching the congestion threshold for the server.
- With the minimal three-segment exchange there is only one RTT that can be measured in each direction. Plus the client's measured RTT includes the server's processing time. This means the smoothed RTT value and its variance also must be cached for the server, similar to what we described in Section 21.9.

The appealing feature of T/TCP is that it is a minimal set of changes to an existing protocol but allows backward compatibility with existing implementations. It also takes advantage of existing engineering features of TCP (dynamic timeout and retransmission, congestion avoidance, etc.) instead of forcing the application to deal with these issues.

An alternative transaction protocol is VMTP, the Versatile Message Transaction Protocol. It is described in RFC 1045 [Cheriton 1988]. Unlike T/TCP, which is a small set of extensions to an existing protocol, VMTP is a complete transport layer that uses IP. VMTP handles error detection, retransmission, and duplicate suppression. It also supports multicast communication.

24.8 TCP Performance

Published numbers in the mid-1980s showed TCP throughput on an Ethernet to be around 100,000 to 200,000 bytes per second. (Section 17.5 of [Stevens 1990] gives these references.) A lot has changed since then. It is now common for off-the-shelf hardware (workstations and faster personal computers) to deliver 800,000 bytes or more per second.

It is a worthwhile exercise to calculate the theoretical maximum throughput we could see with TCP on a 10 Mbits/sec Ethernet [Warnock 1991]. We show the basics for this calculation in Figure 24.9. This figure shows the total number of bytes exchanged for a full-sized data segment and an ACK.

Field	Data #bytes	ACK #bytes
Ethernet preamble	8	8
Ethernet destination address	6	6
Ethernet source address	6	6
Ethernet type field	2	2
IP header	20	20
TCP header	20	20
user data	1460	0
pad (to Ethernet minimum)	0	6
Ethernet CRC	4	4
interpacket gap (9.6 microsec)	12	12
total	1538	84

Figure 24.9 Field sizes for Ethernet theoretical maximum throughput calculation.

We must account for all the overhead: the preamble, the PAD bytes that are added to the acknowledgment, the CRC, and the minimum interpacket gap (9.6 microseconds, which equals 12 bytes at 10 Mbits/sec).

We first assume the sender transmits two back-to-back full-sized data segments, and then the receiver sends an ACK for these two segments. The maximum throughput (user data) is then

$$throughput = \frac{2 \times 1460 \text{ bytes}}{2 \times 1538 + 84 \text{ bytes}} \times \frac{10,000,000 \text{ bits/sec}}{8 \text{ bits/byte}} = 1,155,063 \text{ bytes/sec}$$

If the TCP window is opened to its maximum size (65535, not using the window scale option), this allows a window of 44 1460-byte segments. If the receiver sends an ACK every 22nd segment the calculation becomes

$$throughput = \frac{22 \times 1460 \text{ bytes}}{22 \times 1538 + 84 \text{ bytes}} \times \frac{10,000,000 \text{ bits/sec}}{8 \text{ bits/byte}} = 1,183,667 \text{ bytes/sec}$$

This is the theoretical limit, and makes certain assumptions: an ACK sent by the receiver doesn't collide on the Ethernet with one of the sender's segments; the sender can transmit two segments with the minimum Ethernet spacing; and the receiver can generate the ACK within the minimum Ethernet spacing. Despite the optimism in these numbers, [Warnock 1991] measured a sustained rate of 1,075,000 bytes/sec on an Ethernet, with a standard multiuser workstation (albeit a fast workstation), which is within 90% of the theoretical value.

Moving to faster networks, such as FDDI (100 Mbits/sec), [Schryver 1993] indicates that three commercial vendors have demonstrated TCP over FDDI between 80 and 98 Mbits/sec. When even greater bandwidth is available, [Borman 1992] reports up to 781 Mbits/sec between two Cray Y-MP computers over an 800 Mbits/sec HIPPI channel, and 907 Mbits/sec between two processes using the loopback interface on a Cray Y-MP.

The following practical limits apply for any real-world scenario [Borman 1991].

1. You can't run any faster than the speed of the slowest link.
2. You can't go any faster than the memory bandwidth of the slowest machine. This assumes your implementation makes a single pass over the data. If not (i.e., your implementation makes one pass over the data to copy it from user space into the kernel, then another pass over the data to calculate the TCP checksum), you'll run even slower. [Dalton et al. 1993] describe performance improvements to the standard Berkeley sources that reduce the number of data copies to one. [Partridge and Pink 1993] applied the same "copy-and-checksum" change to UDP, along with other performance improvements, and improved UDP performance by about 30%.
3. You can't go any faster than the window size offered by the receiver, divided by the round-trip time. (This is our bandwidth-delay product equation, using the window size as the bandwidth-delay product, and solving for the bandwidth.) If we use the maximum window scale factor of 14 from Section 24.4, we have a window size of 1.073 Gbytes, so this divided by the RTT is the bandwidth limit.

The bottom line in all these numbers is that the real upper limit on how fast TCP can run is determined by the size of the TCP window and the speed of light. As concluded by [Partridge and Pink 1993], many protocol performance problems are implementation deficiencies rather than inherent protocol limits.

24.9 Summary

This chapter has looked at five new TCP features: path MTU discovery, window scale option, timestamp option, protection against wrapped sequence numbers, and improved transactional processing using TCP. We saw that the middle three features are required for optimal performance on long fat pipes—networks with a large bandwidth-delay product.

Path MTU discovery allows TCP to use windows larger than the default of 536 for nonlocal connections, when the path MTU is larger. This can improve performance.

The window scale option takes the maximum TCP window size from 65535 bytes to just over 1 gigabyte. The timestamp option lets more segments be accurately timed, and also lets the receiver provide protection against wrapped sequence numbers (PAWS). This is essential for high-speed connections. These new TCP options are negotiated at connection establishment, and ignored by older systems that don't understand them, allowing newer systems to interoperate with older systems.

The TCP extensions for transactions, T/TCP, allow a client–server request–reply sequence to be completed using only three segments in the usual case. It avoids the three-way handshake and shortens the TIME_WAIT state by caching a small amount of information for each host with which it has established a connection. It also overloads the data segments with the SYN and FIN flags.

We finished the chapter with a look at TCP performance, since there is still much inaccurate folklore about how fast TCP can run. For a well-tuned implementation using the newer features described in this chapter, TCP performance is limited only by the maximum 1-gigabyte window and the speed of light (i.e., the round-trip time).

Exercises

24.1 What does it mean when a system sends an initial SYN segment with a window scale factor of 0?

24.2 If the host `bsdi` in Figure 24.7 supported the window scale option, what is the expected value of the 16-bit window size field in the TCP header from `vangogh` in segment 3? Similarly, if the option were in use for the second connection in that figure, what would be the advertised window in segment 13?

24.3 Instead of fixing the window scale factor when the connection is established, could the window scale option have been defined to also appear when the scaling factor changes?

24.4 At what data rate does sequence number wrap become a problem, assuming an MSL of 2 minutes?

24.5 PAWS is defined to operate within a single connection only. What modifications would have to be made to TCP to use PAWS as a replacement for the 2MSL wait (the TIME_WAIT state)?

24.6 In our example at the end of Section 24.4, why did our `sock` program output the size of the receive buffer before the line that followed (with the IP addresses and port numbers)?

24.7 Redo the calculations of the *throughput* in Section 24.8 assuming an MSS of 1024.

24.8 How does the timestamp option affect Karn's algorithm (Section 21.3)?

24.9 If TCP sends data with the SYN segment that's generated by an active open (without using the extensions we described in Section 24.7), what does the receiving TCP do with the data?

24.10 In Section 24.7 we said that without the T/TCP extensions, even if the active open is sent with data and a FIN, the client delay in receiving the server's response is still twice the RTT plus SPT. Show the segments to account for this.

24.11 Redo Exercise 18.14 assuming T/TCP support and the minimum *RTO* supported by Berkeley-derived systems of one-half second.

24.12 If we implement T/TCP and measure the transaction time between two hosts, what can we compare it to, to determine its efficiency?

25

SNMP: Simple Network Management Protocol

25.1 Introduction

As the number of networks within an organization grows, along with the diversity of systems comprising this internet (routers from various vendors, hosts with embedded router functionality, terminal servers, etc.), managing all these systems within a coherent framework becomes important. This chapter looks at the standards used within the Internet protocol suite for network management.

Network management of a TCP/IP internet consists of *network management stations* (managers) communicating with *network elements*. The network elements can be anything that runs the TCP/IP protocol suite: hosts, routers, X terminals, terminal servers, printers, and so on. The software in the network element that runs the management software is called the *agent*. Management stations are normally workstations with color monitors that graphically display relevant facts about the elements being monitored (which links are up and down, volume of traffic across various links over time, etc.).

The communication can be two way: the manager asking the agent for a specific value ("how many ICMP port unreachables have you generated?"), or the agent telling the manager that something important happened ("an attached interface has gone down"). Also, the manager should be able to set variables in the agent ("change the value of the default IP TTL to 64"), in addition to reading variables from the agent.

TCP/IP network management consists of three pieces.

1. A *Management Information Base* (MIB) that specifies what variables the network elements maintain (the information that can be queried and set by the manager). RFC 1213 [McCloghrie and Rose 1991] defines the second version of this, called MIB-II.

2. A set of common structures and an identification scheme used to reference the variables in the MIB. This is called the *Structure of Management Information* (SMI) and is specified in RFC 1155 [Rose and McCloghrie 1990]. For example, the SMI specifies that a `Counter` is a nonnegative integer that counts from 0 through 4,294,967,295 and then wraps around to 0.
3. The protocol between the manager and the element, called the *Simple Network Management Protocol* (SNMP). RFC 1157 [Case et al. 1990] specifies the protocol. This details the format of the packets exchanged. Although a wide variety of transport protocols could be used, UDP is normally used with SNMP.

These RFCs define what is now called SNMPv1, or just SNMP, which is the topic of this chapter. During 1993 additional RFCs were published specifying SNMP Version 2 (SNMPv2), which we describe in Section 25.12.

Our approach to SNMP in this chapter is to describe the protocol between the manager and the agent first, and then look at the data types for the variables maintained by the agent. We describe the database of information maintained by the agent (the MIB), looking at the groups that we've described in this text: IP, UDP, TCP, and so on. We show examples at each point along the way, tying network management back to the protocol concepts from earlier chapters.

25.2 Protocol

SNMP defines only five types of messages that are exchanged between the manager and agent.

1. Fetch the value of one or more variables: the `get-request` operator.
2. Fetch the next variable after one or more specified variables: the `get-next-request` operator. (We describe what we mean by "next" later in this chapter.)
3. Set the value of one or more variables: the `set-request` operator.
4. Return the value of one or more variables: the `get-response` operator. This is the message returned by the agent to the manager in response to the `get-request`, `get-next-request`, and `set-request` operators.
5. Notify the manager when something happens on the agent: the `trap` operator.

The first three messages are sent from the manager to the agent, and the last two are from the agent to the manager. (We'll refer to the first three as the `get`, `get-next`, and `set` operators.) Figure 25.1 summarizes these five operators.

Since four of the five SNMP messages are simple request–reply protocols (the manager sends a request, the agent sends back a reply) SNMP uses UDP. This means that a request from the manager may not arrive at the agent, and the agent's reply may not make it back to the manager. The manager probably wants to implement a timeout and retransmission.

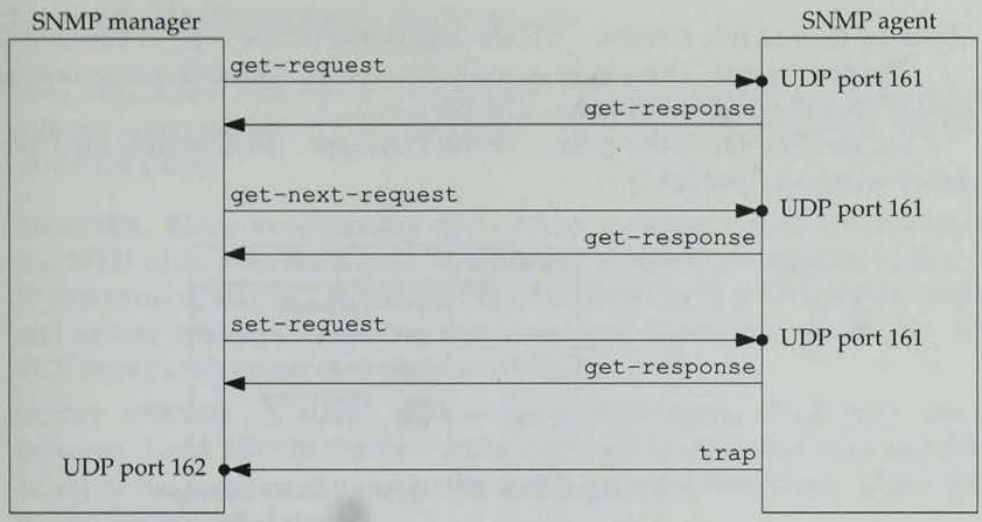


Figure 25.1 Summary of the five SNMP operators.

The manager sends its three requests to UDP port 161. The agent sends traps to UDP port 162. By using two different port numbers, a single system can easily run both a manager and an agent. (See Exercise 25.1.)

Figure 25.2 shows the format of the five SNMP messages, encapsulated in a UDP datagram.

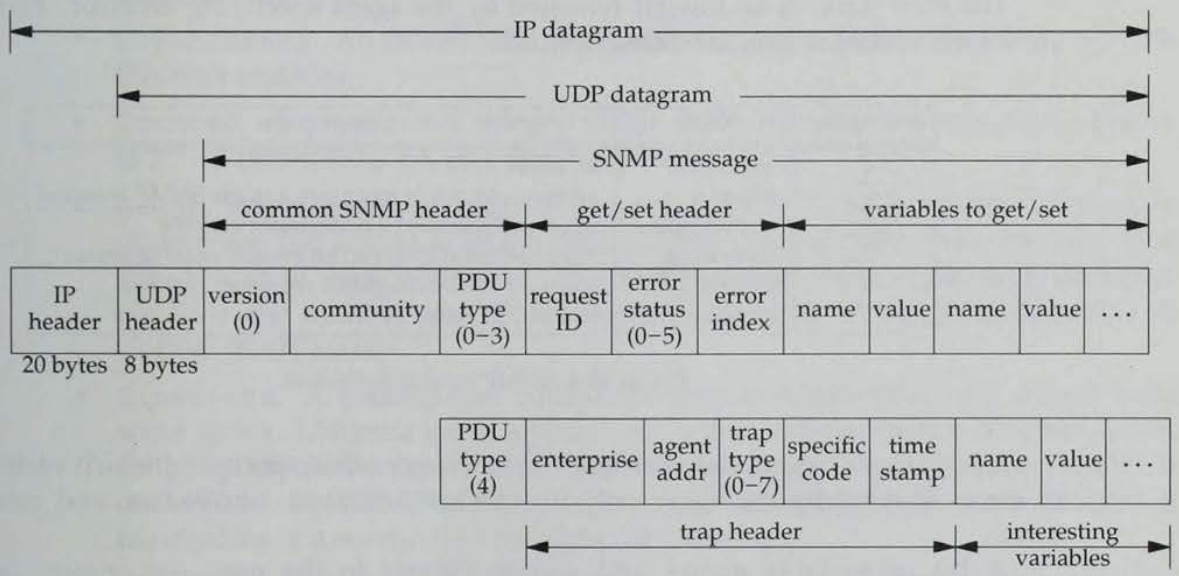


Figure 25.2 Format of the five SNMP messages.

In this figure we specify the size in bytes of the IP and UDP headers only. This is because the encoding used for the SNMP message—called ASN.1 and BER, which we

describe later in this chapter—varies depending on the type of variable and its value.

The *version* is 0. This value is really the version number minus one, as the version of SNMP that we describe is called SNMPv1.

Figure 25.3 shows the values for the *PDU type*. (PDU stands for *Protocol Data Unit*, a fancy word for "packet.")

PDU type	Name
0	`get-request`
1	`get-next-request`
2	`set-request`
3	`get-response`
4	`trap`

Figure 25.3 PDU types for SNMP messages.

The *community* is a character string that is a cleartext password between the manager and agent. A common value is the 6-character string `public`.

For the `get`, `get-next`, and `set` operators, the *request ID* is set by the manager, and returned by the agent in the `get-response` message. We've seen this type of variable with other UDP applications. (Recall the DNS *identification* field in Figure 14.3, and the *transaction ID* field in Figure 16.2.) It lets the client (the manager in this case) match the responses from the server (the agent) to the queries that the client issued. This field also allows the manager to issue multiple requests to one or more agents, and then be able to sort out the returned replies.

The *error status* is an integer returned by the agent specifying an error. Figure 25.4 shows the values, names, and descriptions.

error status	Name	Description
0	`noError`	all is OK
1	`tooBig`	agent could not fit reply into a single SNMP message
2	`noSuchName`	operation specified a nonexistent variable
3	`badValue`	a set operation specified an invalid value or syntax
4	`readOnly`	manager tried to modify a read-only variable
5	`genErr`	some other error

Figure 25.4 SNMP error status values.

If an error occurred, the *error index* is an integer offset specifying which variable was in error. It is set by the agent only for the `noSuchName`, `badValue`, and `readOnly` errors.

A list of variable names and values follows in the `get`, `get-next`, and `set` requests. The value portion is ignored for the `get` and `get-next` operators.

For the `trap` operator (a *PDU type* of 4), the format of the SNMP message changes. We describe the fields in the trap header when we describe this operator in Section 25.10.

25.3 Structure of Management Information

SNMP uses only a few different types of data. In this section we'll look at those data types, without worrying about how the data is actually encoded (that is, the bit pattern used to store the data).

- `INTEGER`. Some variables are declared as an integer with no restrictions (e.g., the MTU of an interface), some are defined as taking on specific values (e.g., the IP forwarding flag is 1 if forwarding is enabled or 2 if forwarding is disabled), and others are defined with a minimum and maximum value (e.g., UDP and TCP port numbers are between 0 and 65535).
- `OCTET STRING`. A string of 0 or more 8-bit bytes. Each byte has a value between 0 and 255. In the BER encoding used for this data type and the next, a count of the number of bytes in the string precedes the string. These strings are not null-terminated strings.
- `DisplayString`. A string of 0 or more 8-bit bytes, but each byte must be a character from the NVT ASCII set (Section 26.4). All variables of this type in the MIB-II must contain no more than 255 characters. (A 0-length string is OK.)
- `OBJECT IDENTIFIER`. We describe these in the next section.
- `NULL`. This indicates that the corresponding variable has no value. It is used, for example, as the value of all the variables in a `get` or `get-next` request, since the values are being queried, not set.
- `IpAddress`. An `OCTET STRING` of length 4, with 1 byte for each byte of the IP address.
- `PhysAddress`. An `OCTET STRING` specifying a physical address (e.g., a 6-byte Ethernet address).
- `Counter`. A nonnegative integer whose value increases monotonically from 0 to $2^{32} - 1$ (4,294,967,295), and then wraps back to 0.
- `Gauge`. A nonnegative integer between 0 and $2^{32} - 1$, whose value can increase or decrease, but latches at its maximum value. That is, if the value increments to $2^{32} - 1$, it stays there until reset. The MIB variable `tcpCurrEstab` is an example: it is the number of TCP connections currently in the ESTABLISHED or CLOSE_WAIT state.
- `TimeTicks`. A counter that counts the time in hundredths of a second since some epoch. Different variables can specify this counter from a different epoch, so the epoch used for each variable of this type is specified when the variable is declared in the MIB. For example, the variable `sysUpTime` is the number of hundredths of a second that the agent has been up.
- `SEQUENCE`. This is similar to a structure in the C programming language. For example, we'll see that the MIB defines a `SEQUENCE` named `UdpEntry` containing information about an agent's active UDP end points. (By "active" we mean ports currently in use by an application.) Two entries are in the structure:

1. `udpLocalAddress`, of type `IpAddress`, containing the local IP address.
2. `udpLocalPort`, of type `INTEGER`, in the range 0 through 65535, specifying the local port number.

- `SEQUENCE OF`. This is the definition of a vector, with all elements having the same data type. If each element has a simple data type, such as an integer, then we have a simple vector (a one-dimensional array). But we'll see that SNMP uses this data type with each element of the vector being a `SEQUENCE` (structure). We can then think of it as a two-dimensional array or table.

  For example, the UDP listener table is named `udpTable` and it is a `SEQUENCE OF` the 2-element `SEQUENCE` (structure) `UdpEntry` that we just described. Figure 25.5 shows this two-dimensional array.

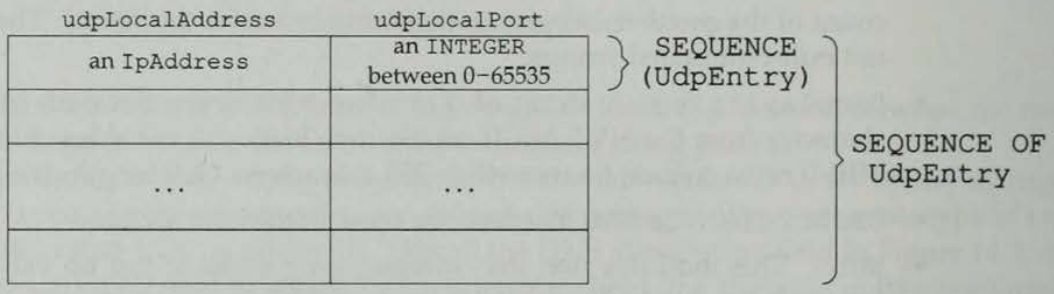


Figure 25.5 UDP listener table (`udpTable`) as a two-dimensional array in SNMP.

The number of rows in these tables is not specified by SNMP, but we'll see that a manager using the `get-next` operator (Section 25.7) can determine when the final row of a table has been returned. Also, in Section 25.6 we'll see how the manager specifies which row of a table it wants to get or set.

25.4 Object Identifiers

An *object identifier* is a data type specifying an authoritatively named object. By "authoritative" we mean that these identifiers are not assigned randomly, but are allocated by some organization that has responsibility for a group of identifiers.

An object identifier is a sequence of integers separated by decimal points. These integers traverse a tree structure, similar to the DNS (Figure 14.1) or a Unix filesystem. There is an unnamed root at the top of the tree where the object identifiers start. (This is the same direction of tree traversal that's used with a Unix filesystem.)

Figure 25.6 shows the structure of this tree when used with SNMP. All variables in the MIB start with the object identifier 1.3.6.1.2.1.

Each node in the tree is also given a textual name. The name corresponding to the object identifier 1.3.6.1.2.1 is `iso.org.dod.internet.mgmt.mib`. These names are for human readability. The *names* of the MIB variables that are in the packets exchanged between the manager and agent (Figure 25.2) are the numeric object identifiers, all of which begin with 1.3.6.1.2.1.

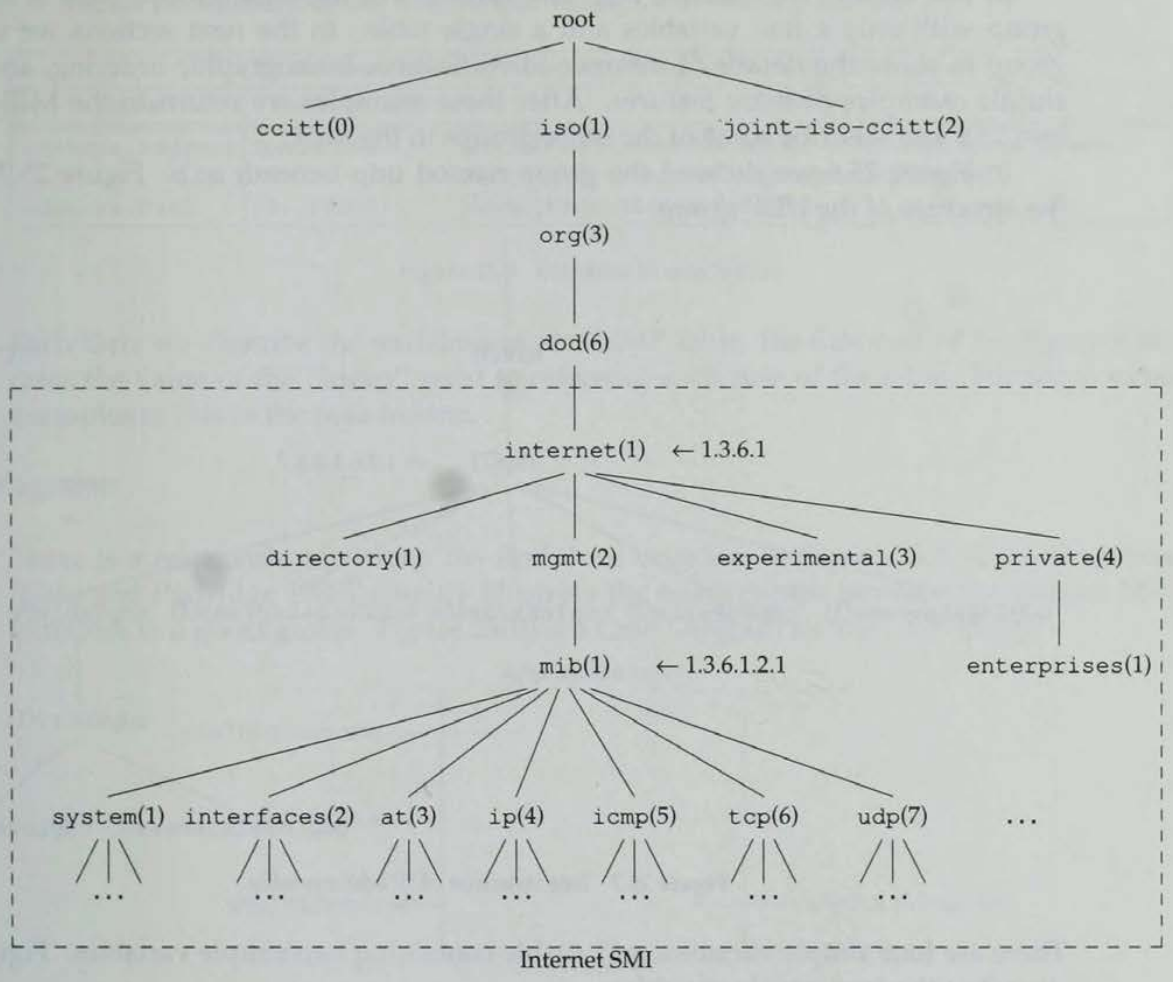


Figure 25.6 Object identifiers in the Management Information Base.

Besides the `mib` object identifiers in Figure 25.6 we also show one named `iso.org.dod.internet.private.enterprises` (1.3.5.1.4.1). This is where vendor-specific MIBs are located. The Assigned Numbers RFC lists around 400 identifiers registered below this node.

25.5 Introduction to the Management Information Base

The *Management Information Base*, or MIB, is the database of information maintained by the agent that the manager can query or set. We describe what's called MIB-II, specified in RFC 1213 [McCloghrie and Rose 1991].

As shown in Figure 25.6, the MIB is divided into groups named `system`, `interfaces`, `at` (address translation), `ip`, and so on.

In this section we describe only the variables in the UDP group. This is a simple group with only a few variables and a single table. In the next sections we use this group to show the details of instance identification, lexicographic ordering, and some simple examples of these features. After these examples we return to the MIB in Section 25.8 and describe some of the other groups in the MIB.

In Figure 25.6 we showed the group named `udp` beneath `mib`. Figure 25.7 shows the structure of the UDP group.

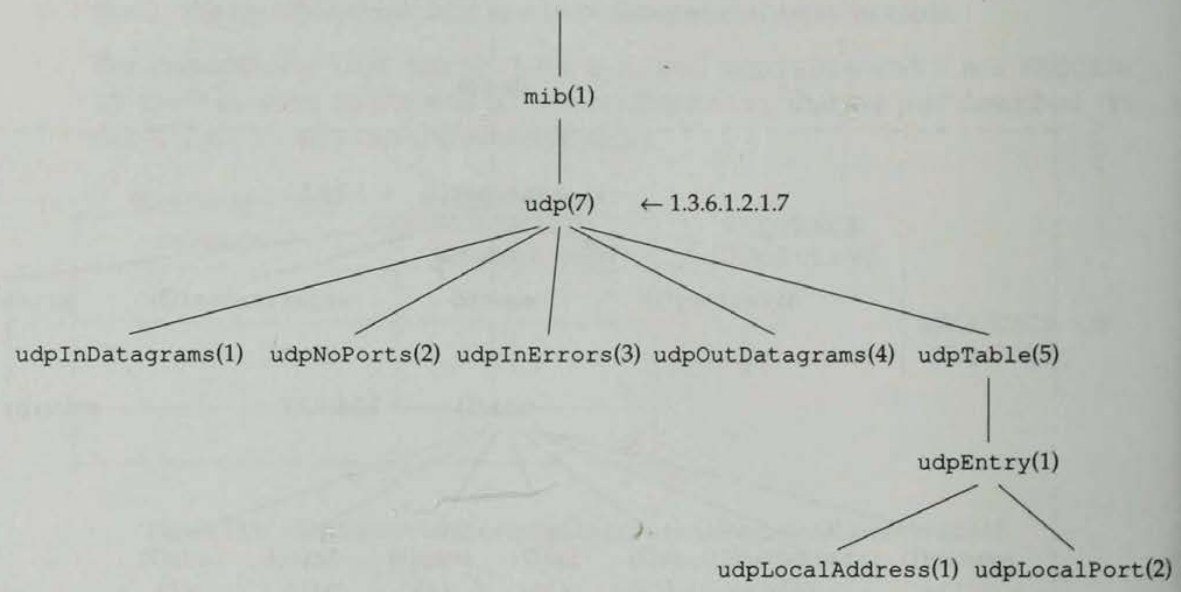


Figure 25.7 Tree structure of IP address table.

There are four simple variables and a table containing two simple variables. Figure 25.8 describes the four simple variables.

Name	Datatype	R/W	Description
`udpInDatagrams`	`Counter`		Number of UDP datagrams delivered to user processes.
`udpNoPorts`	`Counter`		Number of received UDP datagrams for which no application process was at the destination port.
`udpInErrors`	`Counter`		Number of undeliverable UDP datagrams for reasons other than no application at destination port (e.g., UDP checksum error).
`udpOutDatagrams`	`Counter`		Number of UDP datagrams sent.

Figure 25.8 Simple variables in `udp` group.

We'll use this format to describe all the MIB variables in this chapter. The column labeled "R/W" is empty if the variable is read-only, or contains a bullet (•) if the variable is read–write. We always include this column, even if all the variables in a group are read-only (since they are in the udp group) to reiterate that none of the variables can be set by the manager. Also, when the data type is an `INTEGER` with bounds, we specify the lower limit and upper limit, as we do for the UDP port number in the next figure.

Figure 25.9 describes the two simple variables in the udpTable.

UDP listener table, index = < *udpLocalAddress* >.< *udpLocalPort* >			
Name	Datatype	R/W	Description
udpLocalAddress	IpAddress		Local IP address for this listener. 0.0.0.0 indicates the listener is willing to receive datagrams on any interface.
udpLocalPort	[0..65535]		Local port number for this listener.

Figure 25.9 Variables in udpTable.

Each time we describe the variables in an SNMP table, the first row of the figure indicates the value of the "index" used to reference each row of the table. We show some examples of this in the next section.

Case Diagrams

There is a relationship between the first three counters in Figure 25.8. Case Diagrams [Case and Partridge 1989] visually illustrate the relationships between the various MIB variables in a given group. Figure 25.10 is a Case Diagram for the UDP group.

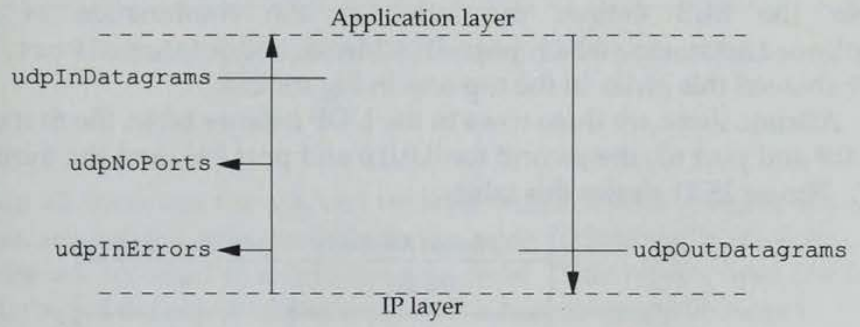


Figure 25.10 Case Diagram for UDP group.

What this diagram shows is that the number of UDP datagrams delivered to applications (udpInDatagrams) is the number of UDP datagrams delivered from IP to UDP, minus udpInErrors, minus udpNoPorts. Also, the number of UDP datagrams delivered to IP (udpOutDatagrams) is the number passed to UDP from the applications. This illustrates that udpInDatagrams does not include udpInErrors or udpNoPorts.

These diagrams were used during the development of the MIB to verify that all data paths for a packet were accounted for. [Rose 1994] shows Case Diagrams for all the groups in the MIB.

25.6 Instance Identification

Every variable in the MIB must be identified when SNMP is referencing it, to fetch or set its value. First, only leaf nodes are referenced. SNMP does not manipulate entire rows or columns of tables. Returning to Figure 25.7, the leaf nodes are the four that we

described in Figure 25.8 and the two in Figure 25.9. `mib`, `udp`, `udpTable`, and `udpEntry` are not leaf nodes.

Simple Variables

Simple variables are referenced by appending ".0" to the variable's object identifier. For example, the counter `udpInDatagrams` from Figure 25.8, whose object identifier is 1.3.6.1.2.1.7.1, is referenced as 1.3.6.1.2.1.7.1.0. The textual name of this reference is `iso.org.dod.internet.mgmt.mib.udp.udpInDatagrams.0`.

Although references to this variable are normally abbreviated as just `udpInDatagrams.0`, we reiterate that the *name* of the variable that appears in the SNMP message (Figure 25.2) is the object identifier 1.3.6.1.2.1.7.1.0.

Tables

Instance identification of table entries is more detailed. Let's return to the UDP listener table (Figure 25.7).

One or more indexes are specified in the MIB for each table. For the UDP listener table, the MIB defines the index as the combination of the two variables `udpLocalAddress`, which is an IP address, and `udpLocalPort`, which is an integer. (We showed this index in the top row in Figure 25.9.)

Assume there are three rows in the UDP listener table: the first row is for IP address 0.0.0.0 and port 67, the second for 0.0.0.0 and port 161, and the third for 0.0.0.0 and port 520. Figure 25.11 shows this table.

udpLocalAddress	udpLocalPort
0.0.0.0	67
0.0.0.0	161
0.0.0.0	520

Figure 25.11 Sample UDP listener table.

This implies that the system is willing to receive UDP datagrams on any interface for ports 67 (BOOTP server), 161 (SNMP), and 520 (RIP). The three rows in the table are referenced as shown in Figure 25.12.

Lexicographic Ordering

There is an implied ordering in the MIB based on the order of the object identifiers. All the entries in MIB tables are lexicographically ordered by their object identifiers. This means the six variables in Figure 25.12 are ordered in the MIB as shown in Figure 25.13. Two key points result from this lexicographic ordering.

Row	Object identifier	Abbreviated name	Value
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67	udpLocalAddress.0.0.0.0.67	0.0.0.0
	1.3.6.1.2.1.7.5.1.2.0.0.0.0.67	udpLocalPort.0.0.0.0.67	67
2	1.3.6.1.2.1.7.5.1.1.0.0.0.0.161	udpLocalAddress.0.0.0.0.161	0.0.0.0
	1.3.6.1.2.1.7.5.1.2.0.0.0.0.161	udpLocalPort.0.0.0.0.161	161
3	1.3.6.1.2.1.7.5.1.1.0.0.0.0.520	udpLocalAddress.0.0.0.0.520	0.0.0.0
	1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalPort.0.0.0.0.520	520

Figure 25.12 Instance identification for rows in UDP listener table.

Column	Object identifier (lexicographically ordered)	Abbreviated name	Value
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67	udpLocalAddress.0.0.0.0.67	0.0.0.0
	1.3.6.1.2.1.7.5.1.1.0.0.0.0.161	udpLocalAddress.0.0.0.0.161	0.0.0.0
	1.3.6.1.2.1.7.5.1.1.0.0.0.0.520	udpLocalAddress.0.0.0.0.520	0.0.0.0
2	1.3.6.1.2.1.7.5.1.2.0.0.0.0.67	udpLocalPort.0.0.0.0.67	67
	1.3.6.1.2.1.7.5.1.2.0.0.0.0.161	udpLocalPort.0.0.0.0.161	161
	1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalPort.0.0.0.0.520	520

Figure 25.13 Lexicographic ordering of UDP listener table.

1. Since all instances for a given variable (`udpLocalAddress`) appear before all instances for the next variable in the table (`udpLocalPort`), this implies that tables are accessed in a column–row order. This results from the lexicographic ordering of the object identifiers, not the human-readable names.
2. The ordering of the rows in a table depends on the values of the indexes for the table. In Figure 25.13, 67 is lexicographically less than 161, which is lexicographically less than 520.

Figure 25.14 shows this column–row order for our sample UDP listener table.

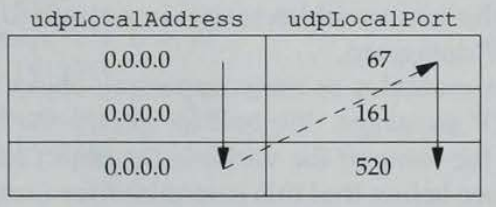


Figure 25.14 UDP listener table, showing column–row ordering.

We'll also see this column–row ordering when we use the `get-next` operator in the next section.

25.7 Simple Examples

In this section we'll show some examples that fetch the values of variables from an SNMP agent. The software used to query the agent is called `snmpi` and is from the ISODE system. Both are described briefly in [Rose 1994].

Simple Variables

We'll query a router for two simple variables from the UDP group:

```
sun % snmpi -a gateway -c secret

snmpi> get udpInDatagrams.0 udpNoPorts.0
udpInDatagrams.0=616168
udpNoPorts.0=33

snmpi> quit
```

The `-a` option identifies the agent we want to communicate with, and the `-c` option specifies the SNMP *community*. It is a password supplied by the client (`snmpi` in this case) and if the server (the agent in the system `gateway`) recognizes the community name, it honors the manager's request. An agent could allow clients within one community read-only access to its variables, and clients in another community read–write access.

The program outputs its `snmpi>` prompt, and we can type commands such as `get`, which translates into an SNMP `get-request` message. When we're done, we type `quit`. (In all further examples we'll remove this final `quit` command.)

Figure 25.15 shows the two lines of `tcpdump` output for this example.

```
1  0.0                  sun.1024 > gateway.161: GetRequest(42)
                        1.3.6.1.2.1.7.1.0 1.3.6.1.2.1.7.2.0

2  0.348875 (0.3489)    gateway.161 > sun.1024: GetResponse(46)
                        1.3.6.1.2.1.7.1.0=616168
                        1.3.6.1.2.1.7.2.0=33
```

Figure 25.15 `tcpdump` output for simple SNMP query.

The request for the two variables is sent in a single UDP datagram, and the response is also a single UDP datagram.

We show the variables as their respective object identifiers, because that is what's sent in the SNMP messages. We had to specify the instance of the two variables as 0. Notice also that the *name* of the variable (its object identifier) is always returned in the response. We'll see below that this is required for the `get-next` operator to work.

`get-next` Operator

The operation of the `get-next` operator is based on the lexicographic ordering of the MIB. We start the following example by asking for the next object identifier after `udp` (without specifying any instance, since this is not a leaf object). This returns the first

object in the UDP group. We then ask for the next entry after this one, and the second entry is returned. We repeat this one more time to get the third entry:

```
sun % snmpi -a gateway -c secret

snmpi> next udp
udpInDatagrams.0=616318

snmpi> next udpInDatagrams.0
udpNoPorts.0=33

snmpi> next udpNoPorts.0
udpInErrors.0=0
```

This example shows why a `get-next` operator must return the name of the variable: we ask the agent for the next variable, and the agent returns its *name* and *value*.

Using the `get-next` operator in this fashion, one could imagine a manager with a loop that starts at the beginning of the MIB and queries the agent for every variable that the agent maintains. Another use of this operator is to iterate through tables.

Table Access

We can reiterate the column–row ordering of tables using our simple query program to step through the entire UDP listener table. We start by asking for the next variable after `udpTable`. Since this is not a leaf object we can't specify an instance, but the `get-next` operator still returns the next object in the table. We then work our way through the table, with the agent returning the next variable, in column–row order:

```
sun % snmpi -a gateway -c secret

snmpi> next udpTable
udpLocalAddress.0.0.0.0.67=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.67
udpLocalAddress.0.0.0.0.161=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.161
udpLocalAddress.0.0.0.0.520=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.520
udpLocalPort.0.0.0.0.67=67

snmpi> next udpLocalPort.0.0.0.0.67
udpLocalPort.0.0.0.0.161=161

snmpi> next udpLocalPort.0.0.0.0.161
udpLocalPort.0.0.0.0.520=520

snmpi> next udpLocalPort.0.0.0.0.520
snmpInPkts.0=59                    we're finished with the UDP listener table
```

We see that the order returned corresponds to Figure 25.14.

How does a manager know when it reaches the end of a table? Since the response to the `get-next` operator contains the name of the next entry in the MIB after the table, the manager can tell when the name changes. In our example the last entry in the UDP listener table is followed by the variable `snmpInPkts`.

25.8 Management Information Base (Continued)

We now return to the description of the MIB. We describe only the following groups: `system` (system identification), `if` (interfaces) , `at` (address translation), `ip`, `icmp`, and `tcp`. Additional groups are defined.

`system` Group

The `system` group is simple; it consists of seven simple variables (i.e., no tables). Figure 25.16 shows their names, data types, and descriptions.

Name	Datatype	R/W	Description
sysDescr	DisplayString		Textual description of entity.
sysObjectID	ObjectID		Vendor's ID within the subtree 1.3.6.1.4.1.
sysUpTime	TimeTicks		Time in hundredths of a second since network management portion of system was rebooted.
sysContact	DisplayString	•	Name of contact person and how to contact them.
sysName	DisplayString	•	Node's fully qualified domain name (FQDN).
sysLocation	DisplayString	•	Physical location of node.
sysServices	[0..127]		Value indicating services provided by node. It is the sum of the layers in the OSI model supported by the node. The following values are added together, depending on the services provided: 0x01 (physical), 0x02 (datalink), 0x04 (internet), 0x08 (end-to-end), 0x40 (application).

Figure 25.16 Simple variables in `system` group.

We can query the router `netb` for some of these variables:

```
sun % snmpi -a netb -c secret

snmpi> get sysDescr.0 sysObjectID.0 sysUpTime.0 sysServices.0
sysDescr.0="Epilogue Technology SNMP agent for Telebit NetBlazer"
sysObjectID.0=1.3.6.1.4.1.12.42.3.1
sysUpTime.0=22 days, 11 hours, 23 minutes, 2 seconds (194178200 timeticks)
sysServices.0=0xc<internet,transport>
```

The system's object identifier is in the `internet.private.enterprises` group (1.3.6.1.4.1) from Figure 25.6 (p. 365). From the Assigned Numbers RFC we can determine that the next object identifier (12) is assigned to the vendor (Epilogue).

We can also see that the `sysServices` variable is the sum of 4 and 8: this element supports the Internet layer (i.e., routing) and the transport layer (i.e., end-to-end).

`interface` Group

Only one simple variable is defined for this group: the number of interfaces on the system, shown in Figure 25.17.

Name	Datatype	R/W	Description
ifNumber	INTEGER		Number of network interfaces on system.

Figure 25.17 Simple variable in `if` group.

This group also defines a table with 22 columns. Each row of the table defines the characteristics for each interface, as shown in Figure 25.18.

Interface table, index = <*IfIndex*>			
Name	Datatype	R/W	Description
ifIndex	INTEGER		Index of interface, between one and ifNumber.
ifDescr	DisplayString		Textual description of interface.
ifType	INTEGER		Type, for example: 6 = Ethernet, 7 = 802.3 Ethernet, 9 = 802.5 token ring, 23 = PPP, 28 = SLIP, and many other values.
ifMtu	INTEGER		MTU of interface.
ifSpeed	Gauge		Speed in bits/sec.
ifPhysAddress	PhysAddress		Physical address, or string of 0 length for interfaces without physical addresses (e.g., serial links).
ifAdminStatus	[1..3]	•	Desired state of interface: 1 = up, 2 = down, 3 = testing.
ifOperStatus	[1..3]		Current state of interface: 1 = up, 2 = down, 3 = testing.
ifLastChange	TimeTicks		Value of sysUpTime when interface entered current operational state.
ifInOctets	Counter		Total number of bytes received, including framing characters.
ifInUcastPkts	Counter		Number of unicast packets delivered to higher layers.
ifInNUcastPkts	Counter		Number of nonunicast (i.e., broadcast or multicast) packets delivered to higher layers.
ifInDiscards	Counter		Number of received packets discarded even though no error in packet (i.e., out of buffers).
ifInErrors	Counter		Number of received packets discarded because of errors.
ifInUnknownProtos	Counter		Number of received packets discarded because of unknown protocol.
ifOutOctets	Counter		Number of bytes transmitted, including framing characters.
ifOutUcastPkts	Counter		Number of unicast packets received from higher layers.
ifOutNUcastPkts	Counter		Number of nonunicast (i.e., broadcast or multicast) packets received from higher layers.
ifOutDiscards	Counter		Number of outbound packets discarded even though no error in packet (i.e., out of buffers).
ifOutErrors	Counter		Number of outbound packets discarded because of errors.
ifOutQLen	Gauge		Number of packets in output queue.
ifSpecific	ObjectID		A reference to MIB definitions specific to this particular type of media.

Figure 25.18 Variables in interface table: `ifTable`.

We can query the host sun for some of these variables for all its interfaces. We expect to find three interfaces, as in Section 3.8, if the SLIP interface is up:

```
sun % snmpi -a sun

snmpi> next ifTable                    first see what index of first interface is
ifIndex.1=1

snmpi> get ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.1="le0"
ifType.1=ethernet-csmacd(6)
ifMtu.1=1500
ifSpeed.1=10000000
ifPhysAddress.1=0x08:00:20:03:f6:42

snmpi> next ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.2="sl0"
ifType.2=propPointToPointSerial(22)
ifMtu.2=552
ifSpeed.2=0
ifPhysAddress.2=0x00:00:00:00:00:00

snmpi> next ifDescr.2 ifType.2 ifMtu.2 ifSpeed.2 ifPhysAddress.2
ifDescr.3="lo0"
ifType.3=softwareLoopback(24)
ifMtu.3=1536
ifSpeed.3=0
ifPhysAddress.3=0x00:00:00:00:00:00
```

We first get five variables for the first interface using the `get` operator, and then get the same five variables for the second interface using the `get-next` operator. The last command gets these same five variables for the third interface, again using the `get-next` command.

The interface type for the SLIP link is reported as proprietary point-to-point serial, not SLIP. Also, the speed of the SLIP link is not reported.

It is critical to understand the relationship between the `get-next` operator and the column–row ordering. When we say `next ifDescr.1` it returns the next row of the table for this variable, not the next variable in the same row. If tables were stored in a row–column order instead, we wouldn't be able to step to the next occurrence of a given variable this way.

`at` Group

The address translation group is mandatory for all systems, but was deprecated by MIB-II. Starting with MIB-II, each network protocol group (e.g., IP) contains its own address translation tables. For IP it is the `ipNetToMediaTable`.

Only a single table with three columns is defined for the `at` group, shown in Figure 25.19.

We can use a new command within the `snmpi` program to dump an entire table. We'll query the router named `kinetics` (which routes between a TCP/IP network and an AppleTalk network) for its entire ARP cache. This output reiterates the lexicographic ordering of the entries in the table:

Address translation table, index = <*atIfIndex*>.1.<*atNetAddress*>			
Name	Datatype	R/W	Description
atIfIndex	INTEGER	•	Interface number: ifIndex.
atPhysAddress	PhysAddress	•	Physical address. Setting this to a string of 0 length invalidates the entry.
atNetAddress	NetworkAddress	•	IP address.

Figure 25.19 Address translation table: atTable.

```
sun % snmpi -a kinetics -c secret dump at

atIfIndex.1.1.140.252.1.4=1
atIfIndex.1.1.140.252.1.22=1
atIfIndex.1.1.140.252.1.183=1
atIfIndex.2.1.140.252.6.4=2
atIfIndex.2.1.140.252.6.6=2

atPhysAddress.1.1.140.252.1.4=0xaa:00:04:00:f4:14
atPhysAddress.1.1.140.252.1.22=0x08:00:20:0f:2d:38
atPhysAddress.1.1.140.252.1.183=0x00:80:ad:03:6a:80
atPhysAddress.2.1.140.252.6.4=0x00:02:16:48
atPhysAddress.2.1.140.252.6.6=0x00:02:3c:48

atNetAddress.1.1.140.252.1.4=140.252.1.4
atNetAddress.1.1.140.252.1.22=140.252.1.22
atNetAddress.1.1.140.252.1.183=140.252.1.183
atNetAddress.2.1.140.252.6.4=140.252.6.4
atNetAddress.2.1.140.252.6.6=140.252.6.6
```

If we watch the packet exchange using tcpdump, when snmpi dumps an entire table it first issues a get-next for the table name (at in this example) to get the first entry. It prints the first entry and issues another get-next. This continues until the entire table has been dumped.

Figure 25.20 shows the arrangement of this table.

atIfIndex	atPhysAddress	atNetAddress
1	0xaa:00:04:00:f4:14	140.252.1.4
1	0x08:00:20:0f:2d:38	140.252.1.22
1	0x00:80:ad:03:6a:80	140.252.1.183
2	0x00:02:16:48	140.252.6.4
2	0x00:02:3c:48	140.252.6.6

Figure 25.20 Example of at table (ARP cache).

The AppleTalk physical addresses on interface number 2 are 32-bit values, not the 48-bit Ethernet addresses to which we're accustomed. Also note that an entry exists for our router (netb at 140.252.1.183), which we expect, since kinetics and netb are on the same Ethernet (140.252.1) and kinetics must use ARP to send the SNMP responses back to us.

ip Group

The ip group defines numerous variables and three tables. Figure 25.21 defines the simple variables.

Name	Datatype	R/W	Description
ipForwarding	[1..2]	•	1 means the system is forwarding IP datagrams, and 2 means it is not.
ipDefaultTTL	INTEGER	•	Default TTL value when transport layer doesn't provide one.
ipInReceives	Counter		Total number of received IP datagrams from all interfaces.
ipInHdrErrors	Counter		Number of IP datagrams discarded because of header errors (e.g., checksum error, version number mismatch, TTL exceeded, etc.).
ipInAddrErrors	Counter		Number of IP datagrams discarded because of incorrect destination address.
ipForwDatagrams	Counter		Number of IP datagrams for which an attempt was made to forward.
ipInUnknownProtos	Counter		Number of locally addressed IP datagrams with an invalid protocol field.
ipInDiscards	Counter		Number of received IP datagrams discarded because of a lack of buffer space.
ipInDelivers	Counter		Number of IP datagrams delivered to appropriate protocol module.
ipOutRequests	Counter		Total number of IP datagrams passed to IP for transmission. Does not include those counted in ipForwDatagrams.
ipOutDiscards	Counter		Number of output IP datagrams discarded because of a lack of buffer space.
ipOutNoRoutes	Counter		Number of IP datagrams discarded because no route could be found.
ipReasmTimeout	INTEGER		Maximum number of seconds that received fragments are held while awaiting reassembly.
ipReasmReqds	Counter		Number of IP fragments received that needed to be reassembled.
ipReasmOKs	Counter		Number of IP datagrams successfully reassembled.
ipReasmFails	Counter		Number of failures by IP reassembly algorithm.
ipFragOKs	Counter		Number of IP datagrams that have been successfully fragmented.
ipFragFails	Counter		Number of IP datagrams that needed to be fragmented but couldn't because the "don't fragment" flag was set.
ipFragCreates	Counter		Number of IP fragments generated by fragmentation.
ipRoutingDiscards	Counter		Number of routing entries chosen to be discarded even though they were valid.

Figure 25.21 Simple variables in ip group.

The first table in the ip group is the IP address table. It contains one row for each IP address on the system. Each row contains five variables, described in Figure 25.22.

IP address table, index = <*ipAdEntAddr*>			
Name	Datatype	R/W	Description
ipAdEntAddr	IpAddress		IP address for this row.
ipAdEntIfIndex	INTEGER		Corresponding interface number: ifIndex.
ipAdEntNetMask	IpAddress		Subnet mask for this IP address.
ipAdEntBcastAddr	[0..1]		Value of least-significant bit of the IP broadcast address. Normally 1.
ipAdEntReasmMaxSize	[0..65535]		Size of largest IP datagram received on this interface that can be reassembled.

Figure 25.22 IP address table: ipAddrTable.

We can query the host sun for its entire IP address table:

```
sun % snmpi -a sun dump ipAddrTable

ipAdEntAddr.127.0.0.1=127.0.0.1
ipAdEntAddr.140.252.1.29=140.252.1.29
ipAdEntAddr.140.252.13.33=140.252.13.33

ipAdEntIfIndex.127.0.0.1=3          loopback interface, lo0
ipAdEntIfIndex.140.252.1.29=2       SLIP interface, sl0
ipAdEntIfIndex.140.252.13.33=1      Ethernet interface, le0

ipAdEntNetMask.127.0.0.1=255.0.0.0
ipAdEntNetMask.140.252.1.29=255.255.255.0
ipAdEntNetMask.140.252.13.33=255.255.255.224

ipAdEntBcastAddr.127.0.0.1=1        all three use one bits for broadcast
ipAdEntBcastAddr.140.252.1.29=1
ipAdEntBcastAddr.140.252.13.33=1

ipAdEntReasmMaxSize.127.0.0.1=65535
ipAdEntReasmMaxSize.140.252.1.29=65535
ipAdEntReasmMaxSize.140.252.13.33=65535
```

The interface numbers can be compared with the output following Figure 25.18, and the IP addresses and subnet masks can be compared with the values output by the ifconfig command in Section 3.8.

The next table, Figure 25.23, is the IP routing table. (Recall our description of routing tables in Section 9.2.) The index used to access each row of the table is the destination IP address.

Figure 25.24 is the IP routing table on the host sun obtained with the dump ipRouteTable command using snmpi. We have deleted all five of the routing metrics, since they are all –1. In the column headings we've also removed the prefix ipRoute from each variable name.

IP routing table, index = < *ipRouteDest* >			
Name	Datatype	R/W	Description
ipRouteDest	IpAddress	•	Destination IP address. A value of 0.0.0.0 indicates a default entry.
ipRouteIfIndex	INTEGER	•	Interface number: ifIndex.
ipRouteMetric1	INTEGER	•	Primary routing metric. The meaning of the metric depends on the routing protocol (ipRouteProto). A value of –1 means it's not used.
ipRouteMetric2	INTEGER	•	Alternative routing metric.
ipRouteMetric3	INTEGER	•	Alternative routing metric.
ipRouteMetric4	INTEGER	•	Alternative routing metric.
ipRouteNextHop	IpAddress	•	IP address of next-hop router.
ipRouteType	INTEGER	•	Route type: 1 = other, 2 = invalidated route, 3 = direct, 4 = indirect.
ipRouteProto	INTEGER		Routing protocol: 1 = other, 4 = ICMP redirect, 8 = RIP, 13 = OSPF, 14 = BGP, and others.
ipRouteAge	INTEGER	•	Number of seconds since route was last updated or determined to be correct.
ipRouteMask	IpAddress	•	Mask to be logically ANDed with destination IP address before being compared with ipRouteDest.
ipRouteMetric5	INTEGER	•	Alternative routing metric.
ipRouteInfo	ObjectID		Reference to MIB definitions specific to this particular routing protocol.

Figure 25.23 IP routing table: ipRouteTable.

Dest	IfIndex	NextHop	Type	Proto	Mask
0.0.0.0	2	140.252.1.183	indirect(4)	other(1)	0.0.0.0
127.0.0.1	3	127.0.0.1	direct(3)	other(1)	255.255.255.255
140.252.1.183	2	140.252.1.29	direct(3)	other(1)	255.255.255.255
140.252.13.32	1	140.252.13.33	direct(3)	other(1)	255.255.0.0
140.252.13.65	1	140.252.13.35	indirect(4)	other(1)	255.255.255.255

Figure 25.24 IP routing table for the router sun.

For comparison, here is the IP routing table in the format output by netstat (which we discussed in Section 9.2). Figure 25.24 is lexicographically ordered, unlike the netstat output:

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use        Interface
140.252.13.65        140.252.13.35        UGH      0      115        le0
127.0.0.1            127.0.0.1            UH       1      1107       lo0
140.252.1.183        140.252.1.29         UH       0      86         sl0
default              140.252.1.183        UG       2      1628       sl0
140.252.13.32        140.252.13.33        U        8      68359      le0
```

The final table in the ip group is the address translation table, Figure 25.25. As we said earlier, the at group is now deprecated, and this IP table replaces it.

IP address translation table, index = < *ipNetToMediaIfIndex* >.< *ipNetToMediaNetAddress* >			
Name	Datatype	R/W	Description
ipNetToMediaIfIndex	INTEGER	•	Corresponding interface: ifIndex.
ipNetToMediaPhysAddress	PhysAddress	•	Physical address.
ipNetToMediaNetAddress	IpAddress	•	IP address.
ipNetToMediaType	[1..4]	•	Type of mapping: 1 = other, 2 = invalidated, 3 = dynamic, 4 = static.

Figure 25.25 IP address translation table: ipNetToMediaTable.

Here is the ARP cache on the system sun:

```
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

and the corresponding SNMP output:

```
sun % snmpi -a sun dump ipNetToMediaTable

ipNetToMediaIfIndex.1.140.252.13.34=1
ipNetToMediaIfIndex.1.140.252.13.35=1
ipNetToMediaPhysAddress.1.140.252.13.34=0x00:00:c0:c2:9b:26
ipNetToMediaPhysAddress.1.140.252.13.35=0x00:00:c0:6f:2d:40
ipNetToMediaNetAddress.1.140.252.13.34=140.252.13.34
ipNetToMediaNetAddress.1.140.252.13.35=140.252.13.35
ipNetToMediaType.1.140.252.13.34=dynamic(3)
ipNetToMediaType.1.140.252.13.35=dynamic(3)
```

icmp Group

The icmp group consists of four general counters (total number of input and output ICMP messages, and number of input and output ICMP messages with errors) and 22 counters for the different ICMP message types: 11 input counters and 11 output counters. These are shown in Figure 25.26.

For the ICMP messages with additional codes (recall from Figure 6.3 that there are 15 different codes for destination unreachable), a separate counter is not maintained by SNMP for each code.

tcp Group

Figure 25.27 describes the simple variables in the tcp group. Many of these refer to the TCP states that we showed in Figure 18.12.

Name	Datatype	R/W	Description
icmpInMsgs	Counter		Total number of received ICMP messages.
icmpInErrors	Counter		Number of received ICMP messages with errors (e.g., invalid ICMP checksum).
icmpInDestUnreachs	Counter		Number of received ICMP destination unreachable message.
icmpInTimeExcds	Counter		Number of received ICMP time exceeded message.
icmpInParmProbs	Counter		Number of received ICMP parameter problem message.
icmpInSrcQuenchs	Counter		Number of received ICMP source quench messages.
icmpInRedirects	Counter		Number of received ICMP redirect messages.
icmpInEchos	Counter		Number of received ICMP echo request messages.
icmpInEchoReps	Counter		Number of received ICMP echo reply messages.
icmpInTimestamps	Counter		Number of received ICMP timestamp request messages.
icmpInTimestampReps	Counter		Number of received ICMP timestamp reply messages.
icmpInAddrMasks	Counter		Number of received ICMP address mask request messages.
icmpInAddrMaskReps	Counter		Number of received ICMP address mask reply messages.
icmpOutMsgs	Counter		Total number of output ICMP messages.
icmpOutErrors	Counter		Number of ICMP messages not sent because of a problem within ICMP (e.g., lack of buffers).
icmpOutDestUnreachs	Counter		Number of ICMP destination unreachable messages sent.
icmpOutTimeExcds	Counter		Number of ICMP time exceeded messages sent.
icmpOutParmProbs	Counter		Number of ICMP parameter problem messages sent.
icmpOutSrcQuenchs	Counter		Number of ICMP source quench messages sent.
icmpOutRedirects	Counter		Number of ICMP redirect messages sent.
icmpOutEchos	Counter		Number of ICMP echo request messages sent.
icmpOutEchoReps	Counter		Number of ICMP echo reply messages sent.
icmpOutTimestamps	Counter		Number of ICMP timestamp requests sent.
icmpOutTimestampReps	Counter		Number of ICMP timestamp reply messages sent.
icmpOutAddrMasks	Counter		Number of ICMP address mask request messages sent.
icmpOutAddrMaskReps	Counter		Number of ICMP address mask reply messages sent.

Figure 25.26 Simple variables in `icmp` group.

We can query some of these variables on the system `sun`:

```
sun % snmpi -a sun

snmpi> get tcpRtoAlgorithm.0 tcpRtoMin.0 tcpRtoMax.0 tcpMaxConn.0
tcpRtoAlgorithm.0=vanj(4)
tcpRtoMin.0=200
tcpRtoMax.0=12800
tcpMaxConn.0=-1
```

This system (SunOS 4.1.3) uses the Van Jacobson retransmission timeout algorithm, uses timeouts between 200 ms and 12.8 seconds, and has no fixed limit on the number of TCP connections. (This upper limit of 12.8 seconds appears wrong, since most implementations use an upper limit of 64 seconds, as we saw in Chapter 21.)

The `tcp` group has a single table, the TCP connection table, shown in Figure 25.28. This contains one row for each connection. Each row contains five variables: the state of the connection, local IP address, local port number, remote IP address, and remote port number.

Name	Datatype	R/W	Description
tcpRtoAlgorithm	INTEGER		Algorithm used to calculate retransmission timeout value: 1 = none of the following, 2 = a constant RTO, 3 = MIL-STD-1778 Appendix B, 4 = Van Jacobson's algorithm.
tcpRtoMin	INTEGER		Minimum retransmission timeout value, in milliseconds.
tcpRtoMax	INTEGER		Maximum retransmission timeout value, in milliseconds.
tcpMaxConn	INTEGER		Maximum number of TCP connections. Value is –1 if dynamic.
tcpActiveOpens	Counter		Number of transitions from CLOSED to SYN_SENT states.
tcpPassiveOpens	Counter		Number of transitions from LISTEN to SYN_RCVD states.
tcpAttemptFails	Counter		Number of transitions from SYN_SENT or SYN_RCVD to CLOSED, plus number of transitions from SYN_RCVD to LISTEN.
tcpEstabResets	Counter		Number of transitions from ESTABLISHED or CLOSE_WAIT states to CLOSED.
tcpCurrEstab	Gauge		Number of connections currently in ESTABLISHED or CLOSE_WAIT states.
tcpInSegs	Counter		Total number of segments received.
tcpOutSegs	Counter		Total number of segments sent, excluding those containing only retransmitted bytes.
tcpRetransSegs	Counter		Total number of retransmitted segments.
tcpInErrs	Counter		Total number of segments received with an error (such as invalid checksum).
tcpOutRsts	Counter		Total number of segments sent with RST flag set.

Figure 25.27 Simple variables in `tcp` group.

index = < *tcpConnLocalAddress* >.< *tcpConnLocalPort* >.< *tcpConnRemAddress* >.< *tcpConnRemPort* >			
Name	Datatype	R/W	Description
tcpConnState	[1..12]	•	State of connection: 1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT_1, 7 = FIN_WAIT_2, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = delete TCB. The only value that the manager can set this variable to is 12 (e.g., immediately terminate the connection).
tcpConnLocalAddress	IpAddress		Local IP address. 0.0.0.0 indicates the listener is willing to accept connections on any interface.
tcpConnLocalPort	[0..65535]		Local port number.
tcpConnRemAddress	IpAddress		Remote IP address.
tcpConnRemPort	[0..65535]		Remote port number.

Figure 25.28 TCP connection table: `tcpConnTable`.

Let's look at this table on the system `sun`. We show only a portion of the table, since there are many servers listening for connections. Before dumping this table two TCP connections were established:

```
sun % rlogin gemini                    IP address of gemini is 140.252.1.11
```

and

```
sun % telnet localhost                 IP address should be 127.0.0.1
```

The only listening server that we show is the FTP server, on port 21:

```
sun % snmpi -a sun dump tcpConnTable

tcpConnState.0.0.0.0.21.0.0.0.0.0=listen(2)
tcpConnState.127.0.0.1.23.127.0.0.1.1415=established(5)
tcpConnState.127.0.0.1.1415.127.0.0.1.23=established(5)
tcpConnState.140.252.1.29.1023.140.252.1.11.513=established(5)

tcpConnLocalAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnLocalAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnLocalAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnLocalAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.29

tcpConnLocalPort.0.0.0.0.21.0.0.0.0.0=21
tcpConnLocalPort.127.0.0.1.23.127.0.0.1.1415=23
tcpConnLocalPort.127.0.0.1.1415.127.0.0.1.23=1415
tcpConnLocalPort.140.252.1.29.1023.140.252.1.11.513=1023

tcpConnRemAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnRemAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnRemAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnRemAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.11

tcpConnRemPort.0.0.0.0.21.0.0.0.0.0=0
tcpConnRemPort.127.0.0.1.23.127.0.0.1.1415=1415
tcpConnRemPort.127.0.0.1.1415.127.0.0.1.23=23
tcpConnRemPort.140.252.1.29.1023.140.252.1.11.513=513
```

For the `rlogin` to `gemini` only one entry appears, since `gemini` is a different host. We only see the client end of the connection (local port 1023), but both ends of the Telnet connection appear (client port 1415 and server port 23), since the connection is through the loopback interface. We can also see that the listening FTP server has a local IP address of 0.0.0.0, indicating it will accept connections on any interface.

25.9 Additional Examples

We now return to some earlier problems we encountered in the text, and use SNMP to understand what's happening.

Interface MTU

Recall our experiment in Section 11.6, in which we tried to determine the MTU of the SLIP link from `netb` to `sun`. We can now use SNMP to obtain this MTU. We first obtain the interface number (`ipRouteIfIndex`) of the SLIP link (140.252.1.29) from the IP routing table. Using this we go into the interface table and fetch the MTU (along with the description and type) of the SLIP link:

```
sun % snmpi -a netb -c secret

snmpi> get ipRouteIfIndex.140.252.1.29
ipRouteIfIndex.140.252.1.29=12
```

```
snmpi> get ifDescr.12 ifType.12 ifMtu.12
ifDescr.12="Telebit NetBlazer dynamic dial virtual interface"
ifType.12=other(1)
ifMtu.12=1500
```

We see that even though the link is a SLIP link, the MTU is set to the Ethernet value of 1500, probably to avoid fragmentation.

Routing Tables

Recall our discussion of address sorting performed by the DNS in Section 14.4 (p. 194). We showed how the first IP address returned by the name server was the one that shared a subnet with the client. We also mentioned that using the other IP address would probably work, but could be less efficient. Let's look at using the alternative IP address and see what happens. We'll use SNMP to look at a routing table entry, and tie together many concepts from earlier chapters dealing with IP routing.

The host `gemini` is multihomed, with two Ethernet interfaces. First let's verify that we can Telnet to both addresses:

```
sun % telnet 140.252.1.11 daytime
Trying 140.252.1.11 ...
Connected to 140.252.1.11.
Escape character is '^]'.
Sat Mar 27 09:37:24 1993
Connection closed by foreign host.

sun % telnet 140.252.3.54 daytime
Trying 140.252.3.54 ...
Connected to 140.252.3.54.
Escape character is '^]'.
Sat Mar 27 09:37:35 1993
Connection closed by foreign host.
```

So there is no connectivity difference between the two addresses. Now we'll use `traceroute` to see if there is a different route for each address:

```
sun % traceroute 140.252.1.11
traceroute to 140.252.1.11 (140.252.1.11), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  299 ms  234 ms  233 ms
 2  gemini (140.252.1.11)  233 ms  228 ms  234 ms

sun % traceroute 140.252.3.54
traceroute to 140.252.3.54 (140.252.3.54), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  245 ms  212 ms  234 ms
 2  swnrt (140.252.1.6)  233 ms  229 ms  234 ms
 3  gemini (140.252.3.54)  234 ms  233 ms  234 ms
```

There is an extra hop if we use the address on subnet 140.252.3. Let's find the reason for the extra hop.

Figure 25.29 shows the arrangement of the systems. We can tell from the `traceroute` output that the host `gemini` and the router `swnrt` are both connected to two networks: 140.252.1 and 140.252.3.

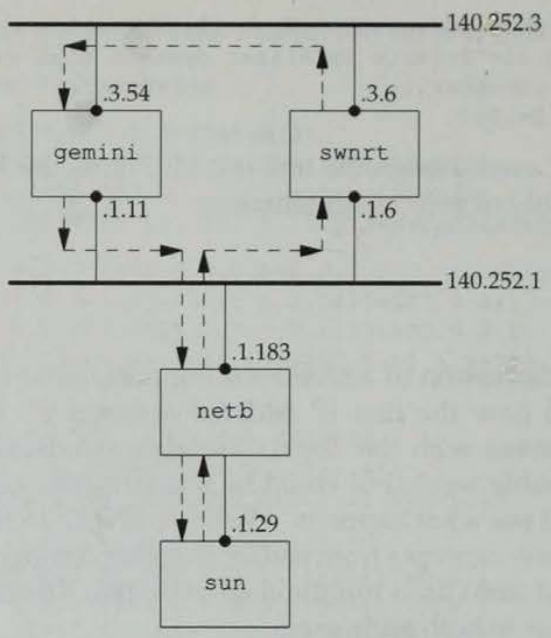


Figure 25.29 Topology of systems being used for example.

Recall in Figure 4.6 that we explained how proxy ARP is used by the router `netb` to make it appear as though `sun` was directly connected to the Ethernet 140.252.1. We've also omitted the modems on the SLIP link between `sun` and `netb`, since they're not relevant to this discussion.

In Figure 25.29 we show the path of the Telnet data using dashed arrows, when the address 140.252.3.54 is specified. How do we know that the return packets go directly from `gemini` to `netb`, and don't go back the way they came? We use our version of `traceroute` with loose source routing from Section 8.5:

```
sun % traceroute -g 140.252.3.54 sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  244 ms  256 ms  234 ms
 2  * * *
 3  gemini (140.252.3.54)  285 ms  227 ms  234 ms
 4  netb (140.252.1.183)  263 ms  259 ms  294 ms
 5  sun (140.252.13.33)  534 ms  498 ms  504 ms
```

When we specify loose source routing, the router `swnrt` never responds. If we look at the earlier output from `traceroute`, without source routing, we see that `swnrt` is indeed the second hop. The reason for the timeouts must be that the router does not generate the ICMP time exceeded errors when the datagram specifies loose source routing. What we are looking for in this `traceroute` output is that the return path from `gemini` (TTLs 3, 4, and 5) goes directly to `netb`, and not through the router `swnrt`.

The question that we need SNMP to answer is what does the routing table entry on `netb` look like for the destination network 140.252.3? It is `netb` that sends the packets to `swnrt` and not directly to `gemini`. We use the `get` command to fetch the value of the next-hop router for this destination:

```
sun % snmpi -a netb -c secret get ipRouteNextHop.140.252.3.0
ipRouteNextHop.140.252.3.0=140.252.1.6
```

This routing table entry tells `netb` to send the packets to `swnrt`, which is what we see happen.

Why does `gemini` send the packets directly back through `netb`? Because on `gemini` the destination address of the return packets is 140.252.1.29, and that network (140.252.1) is a directly connected interface.

What we're seeing in this example is a policy routing decision. The default route to network 140.252.3 is through the router `swnrt` because `gemini` is intended to be a multihomed host, not a router. This is an example of a multihomed host that does not want to be a router.

25.10 Traps

All the examples we've looked at so far in this chapter have been from the manager to the agent. As shown in Figure 25.1, it's also possible for the agent to send a *trap* to the manager, to indicate that something has happened on the agent that the manager might want to know about. Traps are sent to UDP port 162 on the manager.

In Figure 25.2 we showed the format of the trap PDU. We'll go through all the fields in this message when we look at some `tcpdump` output below.

Six specific traps are defined, with a seventh one allowing a vendor to implement an enterprise-specific trap. Figure 25.30 describes the values for the *trap type* in the trap message (Figure 25.2).

trap type	Name	Description
0	`coldStart`	Agent is initializing itself.
1	`warmStart`	Agent is reinitializing itself.
2	`linkDown`	An interface has changed from the up to the down state (Figure 25.18). The first variable in the message identifies the interface.
3	`linkUp`	An interface has changed from the down to the up state (Figure 25.18). The first variable in the message identifies the interface.
4	`authenticationFailure`	A message was received from an SNMP manager with an invalid community.
5	`egpNeighborLoss`	An EGP peer has changed to the down state. The first variable in the messages contains the IP address of the peer.
6	`enterpriseSpecific`	Look in the *specific code* field for information on the trap.

Figure 25.30 Trap types.

We can see some traps using `tcpdump`. We'll start the SNMP agent on the system `sun` and see it generate a `coldStart` trap. (We tell the agent to send traps to the host `bsdi`. Although we're not running a manager on `bsdi` to handle the traps, we can run

tcpdump and see what packets get generated. Recall from Figure 25.1 that a trap is sent from the agent to the manager, but there is no acknowledgment sent by the manager, so we don't need a manager to handle the traps.) We then send a request using the snmpi program, but with an invalid community name. This should generate an authenticationFailure trap. Figure 25.31 shows the output.

```
1  0.0              sun.snmp > bsdi.snmp-trap: C=traps Trap(28)
                    E:unix.1.2.5 [140.252.13.33] coldStart 20

2  18.86 (18.86)    sun.snmp > bsdi.snmp-trap: C=traps Trap(29)
                    E:unix.1.2.5 [140.252.13.33] authenticationFailure 1907
```

Figure 25.31 tcpdump output of traps generated by SNMP agent.

First we notice that both UDP datagrams are from the SNMP agent (port 161, printed as the name snmp) with a destination port of 162 (printed as the name snmp-trap).

The notation C=traps is the community name of the trap message. This is a configuration option with the ISODE SNMP agent being used.

The next notation, Trap(28) in line 1 and Trap(29) in line 2 is the *PDU type* and length.

The next field of output for both lines is E:unix.1.2.5. This is the *enterprise*: the agent's sysObjectID. It falls under the 1.3.6.1.4.1 node of the tree in Figure 25.6 (iso.org.dod.internet.private.enterprises), so this agent's object identifier is 1.3.6.1.4.1.4.1.2.5. Its abbreviated name is unix.agents.fourBSD-isode.5. The final number (5) is the version number of this release of the ISODE agent. This enterprise value identifies the agent software generating the trap.

The next field output by tcpdump is the IP address of the agent (140.252.13.33).

The *trap type* is printed as coldStart on line 1, and authenticationFailure on line 2. These correspond to *trap type* values of 0 and 4, respectively (Figure 25.30). Since these are not enterprise-specific traps, the *specific code* must be 0, and is not printed.

Next comes the *timestamp* field, printed as 20 and 1907. This is a TimeTicks value, representing the number of hundredths of a second since the agent initialized. In the case of the cold start trap, the trap was generated 200 ms after the agent was initialized. The tcpdump output indicates that the second trap occurred 18.86 seconds after the first one, which corresponds to the printed value of 1907 hundredths of a second, minus 200 ms.

Figure 25.2 indicates that a trap message can contain interesting variables that the agents wants to send to the manager, but there aren't any in our examples.

25.11 ASN.1 and BER

The formal specification of SNMP uses *Abstract Syntax Notation 1* (ASN.1) and the actual encoding of the bits in the SNMP messages (Figure 25.2) uses the corresponding *Basic Encoding Rules* (BER). Unlike most texts that describe SNMP, we have purposely left a

discussion of ASN.1 and BER until the end. When they're discussed first, it can confuse the reader and obfuscate the real purpose of SNMP—network management. In this section we only give an overview of these two topics. Chapter 8 of [Rose 1990] covers ASN.1 and BER in detail.

ASN.1 is a formal language for describing data and the properties of the data. It says nothing about how the data is stored or encoded. All the fields in the MIB and the SNMP messages are described using ASN.1. For example, the ASN.1 definition of the data type `IpAddress` from the SMI looks like:

```
IpAddress ::=
    [APPLICATION 0]          -- in network-byte order
        IMPLICIT OCTET STRING (SIZE (4))
```

Similarly, from the MIB we find the following definition of a simple variable:

```
udpNoPorts OBJECT-TYPE
    SYNTAX  Counter
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The total number of received UDP datagrams for which
            there was no application at the destination port."
    ::= { udp 2 }
```

The definition of tables using `SEQUENCE` and `SEQUENCE OF` is more complex.

Given these ASN.1 definitions, there are many ways to encode the data into a stream of bits for transmission. SNMP uses BER. The representation of a small integer, such as 64, requires 3 bytes using BER. One byte says the value is an integer, the next byte says how many bytes are used to store the integer (1), and the final byte contains the binary value.

Fortunately the details of ASN.1 and BER are only important to implementors of SNMP. They are not fundamental to the understanding and use of network management.

25.12 SNMP Version 2

During 1993 11 RFCs were published defining revisions to SNMP. The first of these, RFC 1441 [Case et al. 1993], provides an introduction to SNMP Version 2 (SNMPv2). Two books also describe SNMPv2 [Stallings 1993; Rose 1994]. Two publicly available implementations already exist (see Appendix B.3 of [Rose 1994]), but vendor implementations probably won't be widely available until 1994.

In this section we describe the major differences from SNMPv1 to SNMPv2.

1. A new packet type `get-bulk-request` allows the manager to retrieve large blocks of data efficiently.
2. Another new packet type `inform-request` allows one manager to send information to another manager.

3. Two new MIBs are defined: the SNMPv2 MIB and the SNMPv2-M2M MIB (Manager-to-Manager).
4. SNMPv2 provides security enhancements over SNMPv1. In SNMPv1 the community name passed from the manager to the agent is a cleartext password. SNMPv2 can provide authentication and privacy.

As vendors start to provide SNMPv2-capable agents, management stations will also appear that can handle both. [Routhier 1993] describes extending an implementation of SNMPv1 to support SNMPv2.

25.13 Summary

SNMP is a simple request–reply protocol between an SNMP manager and an SNMP agent. The management information base (MIB) defines the variables that are maintained by the agent, for the manager to query or set. Only a limited number of data types are used to define these variables.

All the variables are identified by object identifiers, a hierarchical naming scheme consisting of long strings of numbers that are normally abbreviated into a simple name, for human readability. A specific instance of a variable is identified by appending an instance to the object identifier.

Many SNMP variables are contained in tables, with a fixed number of columns, but a variable number of rows. Fundamental to SNMP is the identification scheme used to identify each row in a table (when we don't know how many rows are in the table), and the lexicographic ordering (column–row order). The end result, SNMP's `get-next` operator, is basic to any SNMP manager.

We then described the following groups of SNMP variables: system, interface, address translation, IP, ICMP, TCP, and UDP. This was followed by two examples, one to determine the MTU of an interface, and the other to look at the routing table of a router.

We completed the chapter by looking at SNMP traps, a way for the agent to notify the manager that something significant has occurred, and a brief mention of ASN.1 and BER. These latter two topics are probably the most confusing aspects of SNMP, but fortunately their details are needed only by implementors.

Exercises

25.1 We said that using two different ports (161 and 162) allows a system to run both a manager and agent. What would happen if the same port number were used for both?

25.2 How would you list an entire routing table using `get-next`?

26

Telnet and Rlogin: Remote Login

26.1 Introduction

Remote login is one of the most popular Internet applications. Instead of having a hard-wired terminal on each host, we can login to one host and then remote login across the network to any other host (that we have an account on, of course).

Two popular applications provide remote login across TCP/IP internets.

1. Telnet is a standard application that almost every TCP/IP implementation provides. It works between hosts that use different operating systems. Telnet uses option negotiation between the client and server to determine what features each end can provide.
2. Rlogin is from Berkeley Unix and was developed to work between Unix systems only, but it has been ported to other operating systems also.

In this chapter we look at both Telnet and Rlogin. We start with Rlogin because it's simpler.

> Telnet is one of the oldest of the Internet applications, dating back to 1969 on the ARPANET. Its name is actually an acronym that stands for "telecommunications network protocol."

Remote login uses the client–server paradigm. Figure 26.1 shows the typical arrangement of the Telnet client and server. (We could draw a similar picture for an Rlogin client and server.)

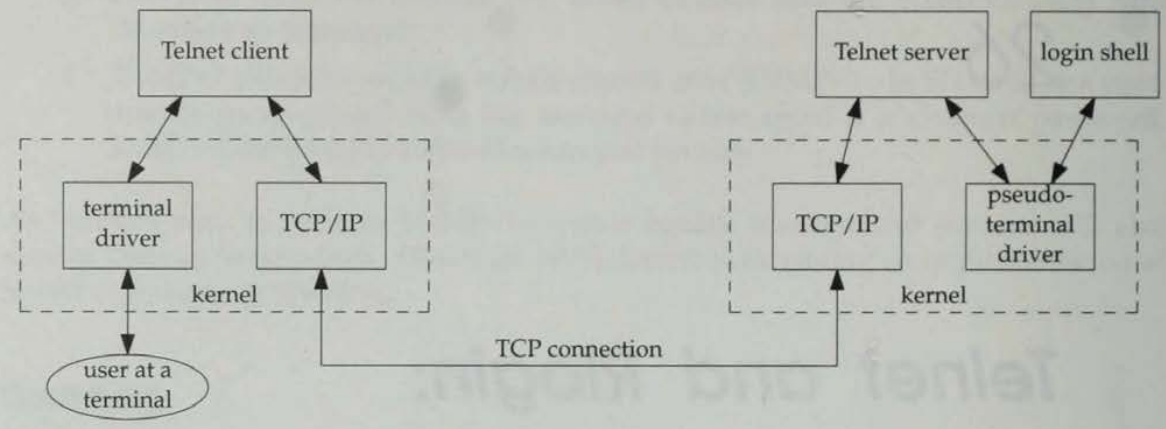


Figure 26.1 Overview of Telnet client–server.

There are numerous points in this figure.

1. The Telnet client interacts with both the user at the terminal and the TCP/IP protocols. Normally everything we type is sent across the TCP connection, and everything received from the connection is output to our terminal.
2. The Telnet server often deals with what's called a *pseudo-terminal* device, at least under Unix systems. This makes it appear to the login shell that's invoked on the server, and to any programs run by the login shell, that they're talking to a terminal device. Some applications, such as full-screen editors, assume they're talking to a terminal device. Indeed, making the login shell on the server think that it's talking to a terminal is often one of the hardest programming aspects involved in writing a remote login server.
3. Only a single TCP connection is used. Since there are times when the Telnet client must talk to the Telnet server (and vice versa) there needs to be some way to delineate commands that are sent across the connection, versus user data. We'll see how both Telnet and Rlogin handle this.
4. We show dashed boxes in Figure 26.1 to note that the terminal and pseudo-terminal drivers, along with the TCP/IP implementation, are normally part of the operating system kernel. The Telnet client and server, however, are often user applications.
5. We show the login shell on the server host to reiterate that we have to login to the server. We must have an account on that system to login to it, using either Telnet or Rlogin.

It is interesting to compare the complexity of Telnet and Rlogin by looking at the number of lines of source code required to implement the client and server for each. Figure 26.2 shows these counts for the standard Telnet and Rlogin client and server, as distributed in different versions from Berkeley (Figure 1.10).

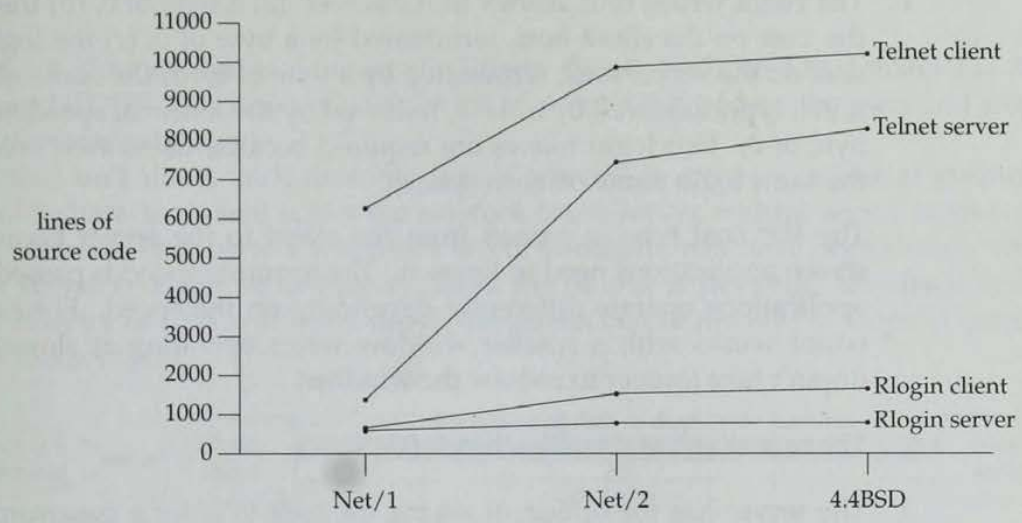


Figure 26.2 Comparison of Telnet/Rlogin/client/server, number of lines of source code.

It is the continuing addition of new options to Telnet that causes its implementation to grow, while Rlogin remains simple and stable.

Remote login is not a high-volume data transfer application. As we've mentioned earlier, lots of small packets are normally exchanged between the two end systems. [Paxson 1993] found that the ratio of bytes sent by the client (the user typing at the terminal) to the number of bytes sent back by the server is about 1:20. This is because we type short commands that often generate lots of output.

26.2 Rlogin Protocol

Rlogin appeared with 4.2BSD and was intended for remote login only between Unix hosts. This makes it a simpler protocol than Telnet, since option negotiation is not required when the operating system on the client and server are known in advance. Over the past few years, Rlogin has also been ported to several non-Unix environments.

RFC 1282 [Kantor 1991] specifies the Rlogin protocol. As with the Routing Information Protocol (RIP) RFC, however, this one was written after Rlogin had been in use for many years. Chapter 15 of [Stevens 1990] describes programming remote login clients and servers, and provides the complete source code for the Rlogin client and server. Chapters 25 and 26 of [Comer and Stevens 1993] provide the implementation details and source code for a Telnet client.

Application Startup

Rlogin uses a single TCP connection between the client and server. After the normal TCP connection establishment is complete, the following application protocol takes place between the client and server.

1. The client writes four strings to the server: (a) a byte of 0, (b) the login name of the user on the client host, terminated by a byte of 0, (c) the login name of the user on the server host, terminated by a byte of 0, (d) the name of the user's terminal type, followed by a slash, followed by the terminal speed, terminated by a byte of 0. Two login names are required because users aren't required to have the same login name on each system.

   The terminal type is passed from the client to the server because many full-screen applications need to know it. The terminal speed is passed because some applications operate differently depending on the speed. For example, the `vi` editor works with a smaller window when operating at slower speeds, so it doesn't take forever to redraw the window.

2. The server responds with a byte of 0.

3. The server has the option of asking the user to enter a password. This is handled as normal data exchange across the Rlogin connection—there is no special protocol. The server sends a string to the client (which the client displays on the terminal), often `Password:`. If the client does not enter a password within some time limit (often 60 seconds), the server closes the connection.

   We can create a file in our home directory on the server (named `.rhosts`) with lines containing a hostname and our username. If we login from the specified host with that username, we are not prompted for a password. Most security texts, such as [Curry 1992], strongly suggest never using this feature because of the security hole it presents.

   If we are prompted by the server for a password, what we type is sent to the server as *cleartext*. Each character of the password that we type is sent as is. Anyone who can read the raw network packets can read the characters of our password. Newer implementations of the Rlogin client, such as 4.4BSD, first try to use Kerberos, which avoids sending cleartext passwords across the network. This requires a compatible server that also supports Kerberos. ([Curry 1992] describes Kerberos in more detail.)

4. The server normally sends a request to the client asking for the terminal's window size (described later).

The client sends 1 byte at a time to the server and all echoing is done by the server. We saw this in Section 19.2. Also, the Nagle algorithm is normally enabled (Section 19.4), causing multiple input bytes to be sent as a single TCP segment across slower networks. The operation is simple: everything typed by the user is sent to the server, and everything sent by the server to the client is displayed on the terminal.

Additional commands exist that can be sent from the client to the server and from the server to the client. Let's first describe the scenarios that require these commands.

Flow Control

By default, flow control is done by the Rlogin client. The client recognizes the ASCII STOP and START characters (Control-S and Control-Q) typed by the user, and stops or starts the terminal output.

If this isn't done, each time we type Control-S to stop the terminal output, the Control-S character is sent across the network to the server, and the server stops writing to the network—but up to a window's worth of output may have been already written by the server and will be displayed before the output is stopped. Hundreds or thousands of bytes of data will scroll down the screen before the output stops. Figure 26.3 shows this scenario.

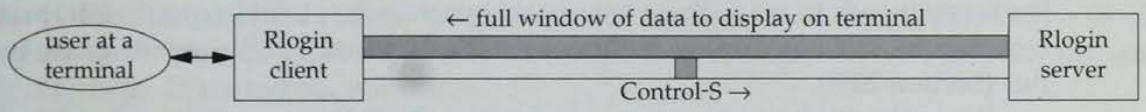


Figure 26.3 Rlogin connection if server performs STOP/START processing.

To an interactive user this delayed response to the Control-S character is bad.

Sometimes, however, the application running on the server needs to interpret each byte of input, and doesn't want the client looking at the input bytes and treating Control-S and Control-Q specially. (The `emacs` editor is an example of an application that uses these two characters for its own commands.) To handle this, the capability is provided for the server to tell the client whether or not to perform flow control.

Client Interrupt

A problem similar to flow control occurs when we type the interrupt key (often DELETE or Control-C), to abort the process currently running on the server. The scenario is similar to what we show in Figure 26.3, with up to one window full of data in the pipe from the server to the client, while the interrupt key makes its way across the connection in the other direction. We want the interrupt key to terminate what's being displayed on the screen as quickly as possible.

In both this case and the flow control scenario, it is rare for the flow of data from the client to the server to be stopped by flow control. This direction contains only characters that we type. Therefore it is not necessary for these special input characters (Control-S or interrupt) to be sent from the client to the server using TCP's urgent mode.

Window Size Changes

With a windowed display we can dynamically change the size of the window while an application is running. Some applications (typically those that manipulate the entire

window, such as a full-screen editor) need to know these changes. Most current Unix systems provide the capability for an application to be told of these window size changes.

With remote login, however, the change in the window size occurs on the client, but the application that needs to be told is running on the server. Some form of notification is required for the Rlogin client to tell the server that the window size has changed, and what the new size is.

Server to Client Commands

We can now summarize the four commands that the Rlogin server can send to the client across the TCP connection. The problem is that only a single TCP connection is used, so the server needs to mark these command bytes so the client knows to interpret them as commands, and not display the bytes on the terminal. TCP's urgent mode is used for this (Section 20.8).

When the server sends a command to the client, the server enters urgent mode with the last byte of urgent data being the command byte from the server. When the client receives the urgent mode notification, it reads from the connection, saving the data until the command byte (the last byte of urgent data) is encountered. The data that's saved by the client can be displayed on the terminal, or discarded, depending on the command. Figure 26.4 describes the four command bytes.

Byte	Description
0x02	Flush output. The client discards all the data received from the server, up through the command byte (the last byte of urgent data). The client also discards any pending terminal output that may be buffered. The server sends this command when it receives the interrupt key from the client.
0x10	The client stops performing flow control.
0x20	The client resumes flow control processing.
0x80	The client responds immediately by sending the current window size to the server, and notifies the server in the future if the window size changes. This command is normally sent by the server immediately after the connection is established.

Figure 26.4 Rlogin commands from the server to the client.

One reason for sending these commands using TCP's urgent mode is that the first command ("flush output") needs to be sent to the client even if the flow of data from the server to the client is stopped by TCP's windowed flow control. This condition—the server's output to the client being flow control stopped—is likely to occur, since processes running on the server can usually generate output faster than the client's terminal can display it. Conversely, it is rare for the flow of data from the client to the server to be flow control stopped, since this direction of data flow contains the characters that we type.

Recall our example in Figure 20.14 where we saw the urgent notification go across the connection even though the window size was 0. (We'll see another example of this in the next section.) The remaining three commands aren't time critical, but they use the same technique for simplicity.

Client to Server Commands

Only one command from the client to the server is currently defined: sending the current window size to the server. Window size changes from the client are not sent to the server unless the client receives the command `0x80` (Figure 26.4) from the server.

Again, since a single TCP connection is used, the client must have some way of marking the commands that it sends across the connection, so that the server doesn't pass them to the application running on the server. The client does this by sending 2 bytes of `0xff` followed by two special flag bytes.

For the window size command, the two flag bytes are each the ASCII character `s`. Following this are four 16-bit values (in network byte order): the number of characters per row (e.g., 25), the number of characters per column (e.g., 80), the number of pixels in the X direction, and the number of pixels in the Y direction. Often the final two 16-bit values are 0, because most applications invoked by the Rlogin server deal with the size of the screen in characters, not pixels.

This form of command that we've described from the client to the server is called *in-band signaling* since the command bytes are sent in the normal stream of data. The bytes used to denote these in-band commands, `0xff`, are chosen because we are unlikely to type keys that generate these bytes. But the Rlogin method is not perfect. If we could generate two consecutive bytes of `0xff` from our keyboard, followed by two ASCII `s`'s, the next 8 bytes we type will be interpreted as window sizes.

The Rlogin commands from the server to the client, which we described in Figure 26.4, are termed *out-of-band signaling* since the technique used is called "out-of-band data" by most APIs. But recall our discussion of TCP's urgent mode in Section 20.8 where we said that urgent mode is not out-of-band data, and the command byte is sent in the normal stream of data, pointed to by the urgent pointer.

Since in-band signaling is used from the client to the server, the server must examine every byte that it receives from the client, looking for two consecutive bytes of `0xff`. But with out-of-band signaling used from the server to the client, the client does not need to examine the data that it receives from the server, until the server enters urgent mode. Even in urgent mode, the client only needs to look at the byte pointed to by the urgent pointer. Since the ratio of bytes from the client to server, versus from the server to client, is about 1:20, it makes sense to use in-band signaling for the low-volume data flow (client to server) and out-of-band signaling for the higher volume data flow (server to client).

Client Escapes

Normally everything we type to the Rlogin client is sent to the server. Occasionally, however, we want to talk directly to the Rlogin client program itself, and not have what we type sent to the server. This is done by typing a tilde (~) as the first character of a line, followed by one of the following four characters:

1. A period terminates the client.
2. The end-of-file character (often Control-D) terminates the client.

3. The job control suspend character (often Control-Z) suspends the client.
4. The job-control delayed-suspend character (often Control-Y) suspends only the client input. Everything we type is now interpreted by whatever program we run on the client host, but anything sent to the Rlogin client by the Rlogin server is output to our terminal. This can be used when we start a long running job on the server and we want to know when it outputs something, but we want to continue running other programs on the client.

The last two commands are supported only if the client Unix system supports job control.

26.3 Rlogin Examples

We'll look at two examples: the first shows the client–server protocol at the start of an Rlogin session, and the second shows what happens when we type our interrupt key to abort a running process on the server that is generating lots of output. In Figure 19.2 we showed the normal flow of data across an Rlogin session.

Initial Client–Server Protocol

Figure 26.5 shows the time line for an Rlogin connection from the host `bsdi` to the server on `svr4`. (We have removed the normal TCP connection establishment, the window advertisements, and the type-of-service information.)

The protocol we described in the previous section can be see in segments 1–9. The client sends a single byte of 0 (segment 1) followed by three strings (segment 3). In this example the three strings are `rstevens` (the login name on the client), `rstevens` (the login name on the server), and `ibmpc3/9600` (the terminal type and speed). The server authenticates this and responds with a byte of 0 (segment 5).

The server then sends the window request command (segment 7). It is sent using TCP's urgent mode and again we see an implementation (SVR4) that uses the older, but more common interpretation, where the urgent pointer specifies the sequence number plus one of the last byte of urgent data. The client responds with 12 bytes of data: 2 bytes of `0xff`, 2 bytes of `s`, and 4 16-bit values.

The next four segments from the server (10, 12, 14, and 16) are the operating system greeting from the server. This is followed by the 7-byte shell prompt on the server: "`svr4 % `" in segment 18.

The data entered by the client is then sent 1 byte at a time, as shown in Figure 19.2. The connection can be closed by either end. If we type a command that causes the shell running on the server to terminate, the server's end does the active close. If we type an escape to the Rlogin client (normally a tilde), followed by a period or our end-of-file character, the client does the active close.

> The client port number in Figure 26.5 is 1023, which is within the range controlled by the IANA (Section 1.9). The Rlogin protocol requires the client to have a port number less than 1024, termed a *reserved port*. On Unix systems, a client cannot obtain a reserved port unless the

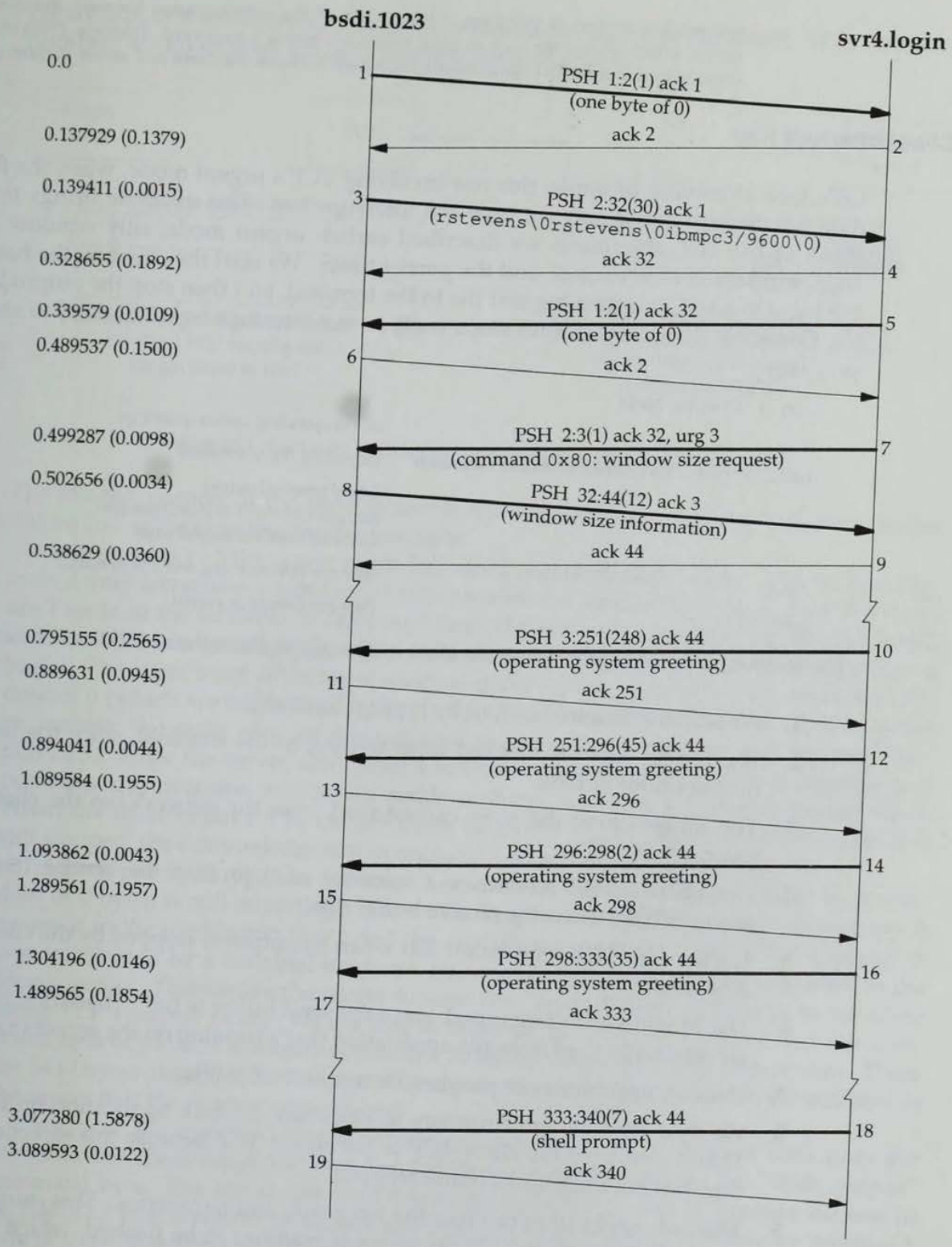


Figure 26.5 Time line for an Rlogin connection.

process has superuser privilege. This is part of the authentication between the client and server, which allows the user to login without entering a password. [Stevens 1990] discusses these reserved ports and the authentication used between the client and server in more detail.

Client Interrupt Key

Let's look at another example, this one involving TCP's urgent mode, when the flow of data has been stopped and we type the interrupt key. This example brings together many of the TCP algorithms we described earlier: urgent mode, silly window avoidance, windowed flow control, and the persist timer. We start the client on the host `sun`. We login to `bsdi`, output a big text file to the terminal, and then stop the output by typing Control-S. When the output stops we type our interrupt key (DELETE) to abort the program:

```
sun % rlogin bsdi
                                        all the operating system greetings
bsdi % cat /usr/share/misc/termcap      output big file to terminal
                                        lots of terminal output
                                        we type Control-S to stop the output,
                                        and wait until the output stops
^?                                      type our interrupt key, and this is echoed
bsdi %                                  then our prompt is output
```

The following points summarize the state of the client, the server, and the connection.

1. We stop the terminal output by typing Control-S.
2. The Rlogin client is blocked from writing to the terminal, since the terminal's output buffer will fill.
3. The Rlogin client therefore cannot read from the network, so the client's TCP receive buffer fills.
4. The client's TCP advertises a window of 0 to stop the sender (the Rlogin server's TCP) when the receive buffer fills.
5. The server's TCP send buffer fills when its output is stopped by the client's window of 0.
6. The Rlogin server is stopped, since the send buffer is full. Therefore, the Rlogin server cannot read from the application that's running on the server (`cat`).
7. The `cat` application stops when its output buffer fills.
8. We then type the interrupt key to terminate the `cat` application on the server. This is sent from the client TCP to the server TCP because this direction of data flow has not been flow-control stopped.
9. The `cat` application receives the interrupt, and terminates. This causes its output buffer (which the Rlogin server is reading) to be flushed, which wakes up the Rlogin server. The Rlogin server then enters urgent mode and sends the "flush output" command (`0x02`) to the client.

Figure 26.6 is a summary of the data flow from the server to the client. (The sequence numbers are taken from the time line that we show next.)

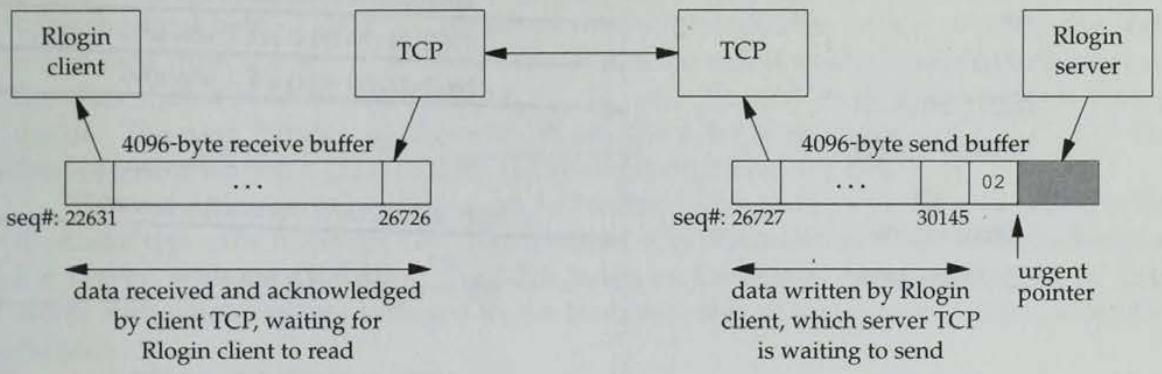


Figure 26.6 Summary of data flow from server to client in Rlogin example.

The shaded portion of the send buffer is the unused portion of the 4096-byte buffer. Figure 26.7 is the time line for this example.

In segments 1–3 the server sends full-sized segments to the client. The ACK in segment 4 only advertises a window of 1024 because the output is stopped: since the client can't write to the terminal, it can't read from the network. Segment 5 is not full sized, and the ACK in segment 6 advertises only the remaining space in the 4096-byte receive buffer. The client must advertise a window of 349 bytes, because if it advertised a window of 0 (which we might expect from silly window avoidance, Section 22.3), it would be moving the right edge of the window to the left, which must not happen (Section 20.3). Since the server can't send a full-sized buffer when it receives segment 6, it performs silly window avoidance, sends nothing, and sets a 5-second persist timer. When the timer expires it sends 349 bytes (segment 7) and since the client's output is still stopped, the acknowledgment in segment 8 advertises a window of 0.

At this point we type our interrupt key and it is transmitted in segment 9. A window of 0 bytes is still advertised. When the Rlogin server receives the interrupt key it passes it to the application (`cat`) and the application terminates. Since the application was terminated by a terminal interrupt, its output is flushed and this is passed to the Rlogin server. This causes the server to send the "flush output" command to the client using TCP's urgent mode. We see this in segment 10. Notice, however, that the command byte of `0x02` is at sequence number 30146 (the urgent pointer minus one). There are 3419 bytes ahead of the command byte (sequence numbers 26727:30145) buffered at the server that the server wants to send.

Segment 10, with the urgent notification, contains the next byte of data from the server to the client (sequence number 26727). It does not contain the "flush output" command byte. The server can send this single byte in segment 10 because we saw in Section 22.2 that a sender with data can always probe a closed window by sending 1 byte of data. The client's TCP responds immediately in segment 11 with a zero window, but the receipt of the urgent notification in segment 10 causes the client's TCP to notify the Rlogin client that the other end of the connection has entered urgent mode.

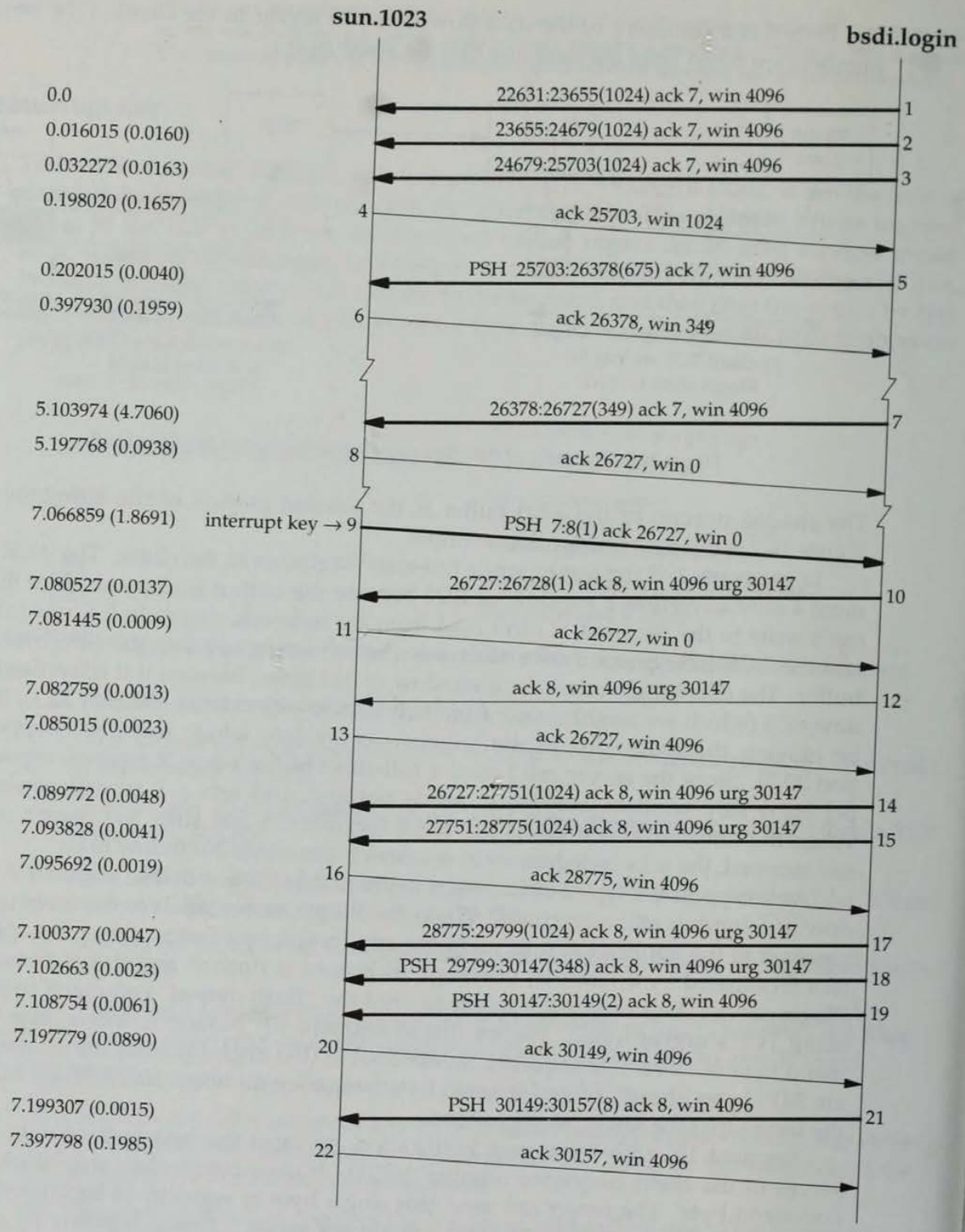


Figure 26.7 Rlogin example when client stops output and then aborts program on server.

Once the Rlogin client receives the urgent notification from its TCP, and starts reading the data that's already waiting for it, the window opens up (segment 13). The data buffered by the server is then sent (segments 14, 15, 17, and 18). The last of these contains the final byte of urgent data (sequence number 30146), which contains the command byte from the server to the client. When the client reads this byte it discards all the data that it read in segments 14, 15, 17, and 18, and flushes its terminal output queue. The next 2 bytes, in segment 19, are the echo of the interrupt key: "^?". The final segment we show (21) contains the shell prompt from the client.

This example shows how data can be buffered at both ends of the connection when the client types the interrupt key. If this action only discarded the 3419 bytes buffered at the server, without discarding the 4096 bytes at the client, these 4096 bytes of data, along with whatever was buffered in the terminal output queue on the client, would be output.

26.4 Telnet Protocol

Telnet was designed to work between any host (i.e., any operating system) and any terminal. Its specification in RFC 854 [Postel and Reynolds 1983a] defines the lowest common denominator terminal, called the *network virtual terminal* (NVT). The NVT is an imaginary device from which both ends of the connection, the client and server, map their real terminal to and from. That is, the client operating system must map whatever type of terminal the user is on to the NVT. The server must then map the NVT into whatever terminal type the server supports.

The NVT is a character device with a keyboard and printer. Data typed by the user on the keyboard is sent to the server, and data received from the server is output to the printer. By default the client echoes what the user types to the printer, but we'll see that options are normally supported to change this.

NVT ASCII

The term *NVT ASCII* refers to the 7-bit U.S. variant of the ASCII character set used throughout the Internet protocol suite. Each 7-bit character is sent as an 8-bit byte, with the high-order bit set to 0.

An end-of-line is transmitted as the 2-character sequence CR (carriage return) followed by an LF (linefeed). We show this as `\r\n`. A carriage return is transmitted as the 2-character sequence CR followed by a NUL (byte of 0). We show this as `\r\0`.

In the following chapters we'll see that FTP, SMTP, Finger, and Whois all use NVT ASCII for client commands and server responses.

Telnet Commands

Telnet uses in-band signaling in both directions. The byte `0xff` (255 decimal) is called IAC, for "interpret as command." The next byte is the command byte. To send the data

byte 255, two consecutive bytes of 255 are sent. (In the previous paragraph we said that the data stream is NVT ASCII, which are 7-bit values, implying a data byte of 255 cannot be sent by Telnet. There is a binary option for Telnet, RFC 856 [Postel and Reynolds 1983b], which we don't discuss, that allows 8-bit data transmission.) Figure 26.8 lists all the Telnet commands.

Name	Code (decimal)	Description
EOF	236	end-of-file
SUSP	237	suspend current process (job control)
ABORT	238	abort process
EOR	239	end of record
SE	240	suboption end
NOP	241	no operation
DM	242	data mark
BRK	243	break
IP	244	interrupt process
AO	245	abort output
AYT	246	are you there?
EC	247	escape character
EL	248	erase line
GA	249	go ahead
SB	250	suboption begin
WILL	251	option negotiation (Figure 26.9)
WONT	252	option negotiation
DO	253	option negotiation
DONT	254	option negotiation
IAC	255	data byte 255

Figure 26.8 Telnet commands, when preceded by IAC (255).

Since many of these commands are rarely used, we describe the important commands when we encounter them in the discussion below and in the examples in the next section.

Option Negotiation

Although Telnet starts with both sides assuming an NVT, the first exchange that normally takes place across a Telnet connection is option negotiation. The option negotiation is symmetric—either side can send a request to the other.

Either side can send one of four different requests for any given option.

1. WILL. The sender wants to enable the option itself.
2. DO. The sender wants the receiver to enable the option.
3. WONT. The sender wants to disable the option itself.
4. DONT. The sender wants the receiver to disable the option.

Since the rules of Telnet allow a side to either accept or reject a request to enable an option (cases 1 and 2 above), but require a side to always honor a request to disable an option (cases 3 and 4 above), these four cases lead to the six scenarios shown in Figure 26.9.

	Sender		Receiver	Description
1.	WILL	→		**sender** wants to **enable** option
		←	DO	receiver says OK
2.	WILL	→		**sender** wants to **enable** option
		←	DONT	receiver says NO
3.	DO	→		sender wants **receiver** to **enable** option
		←	WILL	receiver says OK
4.	DO	→		sender wants **receiver** to **enable** option
		←	WONT	receiver says NO
5.	WONT	→		**sender** wants to **disable** option
		←	DONT	receiver must say OK
6.	DONT	→		sender wants **receiver** to **disable** option
		←	WONT	receiver must say OK

Figure 26.9 Six scenarios for Telnet option negotiation.

Option negotiation requires 3 bytes: the IAC byte, followed by the byte for WILL, DO, WONT, or DONT, followed by an ID byte specifying the option to enable or disable. Currently more than 40 different options can be negotiated. The Assigned Numbers RFC specifies the value for the option byte and the relevant RFC that describes the option. Figure 26.10 shows the option codes that we'll see in this chapter.

Option ID (decimal)	Name	RFC
1	echo	857
3	suppress go ahead	858
5	status	859
6	timing mark	860
24	terminal type	1091
31	window size	1073
32	terminal speed	1079
33	remote flow control	1372
34	linemode	1184
36	environment variables	1408

Figure 26.10 Telnet option codes discussed in the text.

Telnet option negotiation, like most of the Telnet protocol, is intended to be symmetrical. Either end can initiate the negotiation of an option. But remote login is not a symmetrical application. The client performs some tasks, and the server performs others. We'll see as we look at some of the Telnet options that some are intended only for the client (asking to enable linemode, for example), and some are only for the server.

Suboption Negotiation

Some options require more information than just "enable" or "disable." Specifying the terminal type is an example: an ASCII string must be sent by the client identifying the type of terminal. To handle these options, suboption negotiation is defined.

RFC 1091 [VanBokkelen 1989] defines the suboption negotiation for the terminal type. First one side (normally the client) asks to enable the option by sending the 3-byte sequence

<IAC, WILL, 24>

where 24 (decimal) is the option ID for the terminal type option. If the receiver (the client) says OK, its response is

<IAC, DO, 24>

The server then sends

<IAC, SB, 24, 1, IAC, SE>

asking the client for its terminal type. SB is the suboption-begin command. The next byte of 24 identifies that this is a suboption for the terminal type option. (SB is always followed by the option number to which the suboption refers.) The next byte of 1 means "send your terminal type." The suboption-end command must be prefixed by an IAC, just like the SB command. The client responds with the command

<IAC, SB, 24, 0, 'I', 'B', 'M', 'P', 'C', IAC, SE>

if its terminal type is the string `ibmpc`. The fourth byte, 0, means "my terminal type is." (The "official" list of acceptable terminal types is in the Assigned Numbers RFC, but on Unix systems at least, any name that is acceptable to the server is OK. This is normally the terminals supported by either the `termcap` or `terminfo` database.) The terminal type is specified in the Telnet suboption as all uppercase, and normally converted to lowercase by the server.

Half-Duplex, Character at a Time, Line at a Time, or Linemode?

There are four modes of operation for most Telnet clients and servers.

1. Half-duplex.

   This is the default mode, but it is rarely used today. The default NVT is a half-duplex device that requires a GO AHEAD command (GA) from the server before accepting user input. The user input is echoed locally from the NVT keyboard to the NVT printer so that only completed lines are sent from the client to the server.

   While this provides the lowest common denominator for terminal support, it doesn't adequately handle full-duplex terminals communicating with hosts that support full-duplex communications, which is the norm today. RFC 857 [Postel and Reynolds 1983c] defines the ECHO option and RFC 858 [Postel and Reynolds 1983d] defines the SUPPRESS GO AHEAD option. The combination of

these two options provides support for the next mode, character at a time, with remote echo.

2. Character at a time.

   This is what we saw with Rlogin. Each character we type is sent by itself to the server. The server echoes most characters, unless the application on the server turns echoing off.

   The problems with this mode are perceptible delays in echoing across long delay networks and the high volume of network traffic. Nevertheless, we'll see this is the common default for most implementations today.

   We'll see that the way to enter this mode is for the server to have the SUPPRESS GO AHEAD option enabled. This can be negotiated by having the client send a DO SUPPRESS GO AHEAD (asking to enable the option at the server), or the server sending a WILL SUPPRESS GO AHEAD to the client (asking to enable the option itself). The server normally follows this with a WILL ECHO, asking to do the echoing.

3. Line at a time.

   This is often called "kludge line mode," because its implementation comes from reading between the lines in RFC 858. This RFC states that both the ECHO and SUPPRESS GO AHEAD options must be in effect to have character-at-a-time input with remote echo. Kludge line mode takes this to mean that when either of these options is not enabled, Telnet is in a line-at-a-time mode. In the next section we'll see an example of how this mode is negotiated, and how it is disabled when a program that needs to receive every keystroke is run on the server.

4. Linemode.

   We use this term to refer to the real linemode option, defined in RFC 1184 [Borman 1990]. This option is negotiated between the client and server and corrects all the deficiencies in the kludge line mode. Newer implementations support this option.

Figure 26.11 shows the default operating mode between various Telnet clients and servers. The entry "char" means character at a time, "kludge" means the kludge line mode, and "linemode" means the real RFC 1184 linemode.

Client	Server					
	SunOS 4.1.3	Solaris 2.2	SVR4	AIX 3.2.2	BSD/386	4.4BSD
SunOS 4.1.3	char	char	char	char	kludge	kludge
Solaris 2.2	char	char	char	char	kludge	kludge
SVR4	char	char	char	char	kludge	kludge
AIX 3.2.2	char	char	char	char	kludge	kludge
BSD/386	char	char	char	char	linemode	linemode
4.4BSD	char	char	char	char	linemode	linemode

Figure 26.11 Default modes of operation between various Telnet clients and servers.

The only two implementations in this figure that support real linemode are BSD/386 and 4.4BSD. These two servers are also the only ones that attempt to negotiate kludge line mode if real linemode isn't supported by the client. All the clients and servers shown in this figure do support kludge line mode, but they don't select it by default, unless negotiated by the server.

Synch Signal

Telnet defines its *synch signal* as the Data Mark command (DM in Figure 26.8) sent as TCP urgent data. The DM command is the synchronization mark in the data stream that tells the receiver to return to normal processing. It can be sent in either direction across a Telnet connection.

When one end receives notification that the other end has entered urgent mode, it starts reading the data stream, discarding all data other than Telnet commands. The final byte of urgent data is the DM byte. The reason for using TCP's urgent mode is to allow the Telnet commands to be sent across the connection, even if the TCP data flow has been stopped by TCP's flow control.

We'll see examples of Telnet's synch signal in the next section.

Client Escapes

As with the Rlogin client, the Telnet client also lets us talk to it, instead of sending what we type to the server. The normal client escape character is Control-] (control and the right bracket, commonly printed as "^]"). This causes the client to print its prompt, normally "`telnet>` ". There are lots of commands that we can type at this point to change characteristics of the session or to print information. A `help` command is provided by most Unix clients that displays the available commands.

We'll see examples of the client escape, and some of the commands we can issue, in the next section.

26.5 Telnet Examples

We'll now look at Telnet option negotiation, along with the three different modes of operation: character at a time, real linemode, and kludge line mode. We also see what happens when an interactive user aborts a running process on the server with the interrupt key.

Character-at-a-Time Mode

We start with the basic character-at-a-time mode, similar to Rlogin. Each character we type on the terminal is sent by itself to the server, and the server echoes the character. But we'll run a newer client (BSD/386) that tries to enable many newer options, and see them refused by the older server (SVR4).

To see what's negotiated between the client and server we'll enable a client option that displays all the option negotiation, and we'll also run `tcpdump` to obtain a time

line of the packet exchange. Figure 26.12 shows the interactive session.

```
bsdi % telnet                                  invoke client without any command-line options

telnet> toggle options                         tell client to display all the option processing
Will show option processing.

telnet> open svr4                              now establish connection with server
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.

SENT DO SUPPRESS GO AHEAD                      1. (line numbers for discussion that follows)
SENT WILL TERMINAL TYPE                        2.
SENT WILL NAWS                                 3.
SENT WILL TSPEED                               4.
SENT WILL LFLOW                                5.
SENT WILL LINEMODE                             6.
SENT WILL ENVIRON                              7.
SENT DO STATUS                                 8.
RCVD DO TERMINAL TYPE                          9.
RCVD WILL SUPPRESS GO AHEAD                    10.
RCVD DONT NAWS                                 11.
RCVD DONT TSPEED                               12.
RCVD DONT LFLOW                                13.
RCVD DONT LINEMODE                             14.
RCVD DONT ENVIRON                              15.
RCVD WONT STATUS                               16.
RCVD IAC SB TERMINAL-TYPE SEND                 17.
SENT IAC SB TERMINAL-TYPE IS "IBMPC3"          18.
RCVD WILL ECHO                                 19.
SENT DO ECHO                                   20.
RCVD DO ECHO                                   21.
SENT WONT ECHO                                 22.


UNIX(r) System V Release 4.0 (svr4)

RCVD DONT ECHO                                 23.
login: rstevens                                we type our login name
Password:                                      and password, which the server does not echo
                                               operating system greeting is then output ...
                                               then shell prompt
```

Figure 26.12 Initial option negotiation by Telnet client and server.

We've numbered the option negotiation lines that begin with SENT or RCVD, for the discussion that follows.

1. The client initiates the negotiation of the SUPPRESS GO AHEAD option. This option starts with a DO since the GO AHEAD command is normally sent by the server to the client, and the client wants the server to enable the option. (This is confusing since enabling the option disables the GA commands from being sent.) The server OKs this option in line 10.

2. The client wants to send its terminal type as specified in RFC 1091 [VanBokkelen 1989]. This is common with Unix clients. This option starts with a WILL since the client wants to enable the option at its end.
3. NAWS stands for "negotiate about window size" and is defined in RFC 1073 [Waitzman 1988]. If the server agrees (which it doesn't, in line 11), the client then sends a suboption with the number of rows and columns in the terminal window. Additionally, the client will send this suboption at any time later if the window size changes. (This is similar to what we saw with the Rlogin `0x80` command in Figure 26.4.)
4. The TSPEED option lets the sender (normally the client) send its terminal speed, as defined in RFC 1079 [Hedrick 1988b]. If the server agrees (which it doesn't, in line 12), the client then sends a suboption with its transmit speed and receive speed.
5. LFLOW stands for "local flow control," and is defined in RFC 1372 [Hedrick and Borman 1992]. The client sends this option to the server stating that it is willing to enable and disable flow control on command. If the server agrees (which it doesn't in line 13), the server would send a suboption to the client whenever the processing of Control-S and Control-Q needs to switch between the client and server. (This is similar to what we saw with the Rlogin `0x10` and `0x20` commands in Figure 26.4.) As we said in our discussion of Rlogin, the interactive user obtains better response to flow control when it's done by the client, not by the server.
6. LINEMODE is the real linemode that we mentioned in Section 26.4. All the terminal character processing is done by the Telnet client (backspace, erase line, etc.) and complete lines are sent to the server. We'll see an example of it later in this section. This option is refused in line 14.
7. The ENVIRON option lets the client send environment variables to the server, as defined in RFC 1408 [Borman 1993a]. This can automatically propagate variables in the user's environment on the client host to the server. The server refuses this option in line 15. (An environment variable in Unix is often an uppercase name, followed by an equals sign, followed by a string value, but this is only a convention.) By default the BSD/386 Telnet client sends only the two variables `DISPLAY` and `PRINTER`, if they're defined and if the option is enabled. The Telnet user can specify additional environment variables to be sent.
8. The STATUS option (RFC 859 [Postel and Reynolds 1983e]) lets one end ask the other for its perception of the current status of the Telnet options. In this example the client is asking the server to enable the option (DO). If the server agreed (which it doesn't in line 16), the client could ask the server in a suboption to send its status.
9. This is the first response from the server. The server agrees to enable the terminal type option. (Almost every Unix server supports this option.) The client, however, cannot send its terminal type until the server asks for it with a suboption (line 17).

10. The server agrees to suppress sending the GO AHEAD command.
11. The server does not agree to let the client send its window size.
12. The server does not agree to let the client send its terminal speed.
13. The server does not agree to let the client perform flow control.
14. The server does not agree to let the client enable the linemode option.
15. The server does not agree to let the client send environment variables.
16. The server will not send status information.
17. This is a suboption with the server asking the client to send its terminal type.
18. The client sends its terminal type as the 6-character string `IBMPC3`.
19. The server asks the client to let the server perform echoing. This is the first time the server has initiated the negotiation of an option.
20. The client agrees to let the server perform echoing.
21. The server asks the client to perform echoing. This command seems superfluous, given the exchange in the previous two lines, and it is. This is yet another kludge in most Unix Telnet servers to determine if the client is a 4.2BSD host or a later BSD release. If the client responds with WILL ECHO, it is probably an older 4.2BSD host and does not support TCP's urgent mode correctly. (In that case urgent mode won't be used.)
22. The client responds with WONT ECHO, implying it is not a 4.2BSD host.
23. The server responds to the received WONT ECHO with a DONT ECHO.

Figure 26.13 shows the time line for this client–server exchange. (We have removed the connection establishment.)

Segment 1 contains lines 1–8 from Figure 26.12. Each option occupies 3 bytes, for a segment containing 24 bytes. It is the client that starts the option negotiation. This segment shows that multiple Telnet options can appear in a single TCP segment.

Segment 3 is line 9 from Figure 26.12, the DO TERMINAL TYPE command. Segment 5 contains the next eight option responses from the server, lines 10–17 from Figure 26.12. The length of this segment is 27 bytes because lines 10–16 are regular options, each requiring 3 bytes, along with the suboption (line 17), which requires 6 bytes. The 12 bytes in segment 6 correspond to line 18, the client sending the suboption with its terminal type.

Segment 8 (53 bytes) is a combination of two Telnet commands with 47 bytes of data to be output on the terminal. The first 6 bytes are the two commands from the server: WILL ECHO and DO ECHO (lines 19 and 21). The next 47 bytes are:

```
\r\n\r\nUNIX(r) System V Release 4.0 (svr4)\r\n\r\0\r\n\r\0
```

The first 4 bytes produce the two blank lines before the string is output. The 2-byte sequence `\r\n` is considered a newline by Telnet. The 2-byte sequence `\r\0` is considered a carriage return. This segment shows that data and commands can appear in the same segment. The Telnet client and Telnet server must scan every byte they receive, looking for the IAC byte and then processing what follows.