19

TCP Interactive Data Flow

19.1 Introduction

The previous chapter dealt with the establishment and termination of TCP connections. We now examine the transfer of data using TCP.

Studies of TCP traffic, such as [Caceres et al. 1991], usually find that on a packet-count basis about half of all TCP segments contain bulk data (FTP, electronic mail, Usenet news) and the other half contain interactive data (Telnet and Rlogin, for example). On a byte-count basis the ratio is around 90% bulk data and 10% interactive, since bulk data segments tend to be full sized (normally 512 bytes of user data), while interactive data tends to be much smaller. (The above-mentioned study found that 90% of Telnet and Rlogin packets carry less than 10 bytes of user data.)

TCP obviously handles both types of data, but different algorithms come into play for each. In this chapter we'll look at interactive data transfer, using the Rlogin application. We'll see how delayed acknowledgments work and how the Nagle algorithm reduces the number of small packets across wide area networks. The same algorithms apply to Telnet. In the next chapter we'll look at bulk data transfer.

19.2 Interactive Input

Let's look at the flow of data when we type an interactive command on an Rlogin connection. Many newcomers to TCP/IP are surprised to find that each interactive keystroke normally generates a data packet. That is, the keystrokes are sent from the client to the server 1 byte at a time (not one line at a time). Furthermore, Rlogin has the

remote system (the server) echo the characters that we (the client) type. This could generate four segments: (1) the interactive keystroke from the client, (2) an acknowledgment of the keystroke from the server, (3) the echo of the keystroke from the server, and (4) an acknowledgment of the echo from the client. Figure 19.1 shows this flow of data.

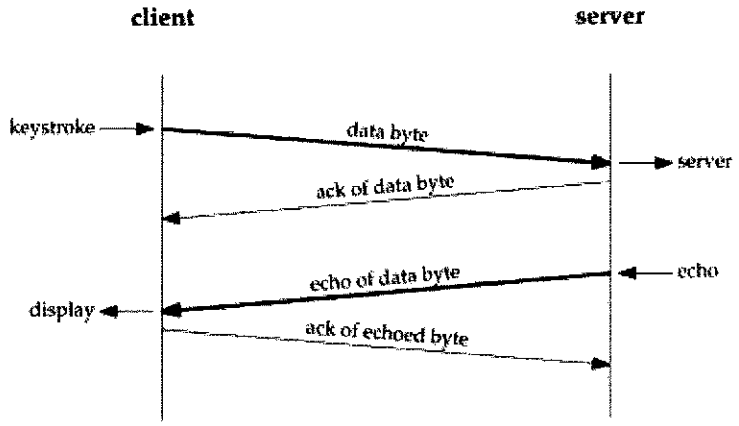

Figure 19.1 One possible way to do remote echo of interactive keystroke.

Normally, however, segments 2 and 3 are combined—the acknowledgment of the keystroke is sent along with the echo. We describe the technique that combines these (called delayed acknowledgments) in the next section.

We purposely use Rlogin for the examples in this chapter because it always sends one character at a time from the client to the server. When we describe Telnet in Chapter 26, we'll see that it has an option that allows lines of input to be sent from the client to the server, which reduces the network load.

Figure 19.2 shows the flow of data when we type the five characters `date\n`. (We do not show the connection establishment and we have removed all the type-of-service output. BSD/386 sets the TOS for an Rlogin connection for minimum delay.) Line 1 sends the character `d` from the client to the server. Line 2 is the acknowledgment of this character and its echo. (This is combining the middle two segments in Figure 19.1.) Line 3 is the acknowledgment of the echoed character. Lines 4–6 correspond to the character `a`, lines 7–9 to the character `t`, and lines 10–12 to the character `e`. The fractional second delays between lines 3–4, 6–7, 9–10, and 12–13 are the human delays between typing each character.

Notice that lines 13–15 are slightly different. One character is sent from the client to the server (the Unix newline character, from our typing the RETURN key) but two characters are echoed. These two characters are a carriage return and linefeed (CR/LF), to move the cursor back to the left and space down one line.

Line 16 is the output of the `date` command from the server. The 30 bytes are composed of the following 28 characters

```
Sat Feb  6 07:52:17 MST 1993
```

```
 1  0.0                bsdi.1023 > svr4.login: P 0:1(1) ack 1 win 4096
 2  0.016497 (0.0165)  svr4.login > bsdi.1023: P 1:2(1) ack 1 win 4096
 3  0.139955 (0.1235)  bsdi.1023 > svr4.login: . ack 2 win 4096

 4  0.458037 (0.3181)  bsdi.1023 > svr4.login: P 1:2(1) ack 2 win 4096
 5  0.474386 (0.0163)  svr4.login > bsdi.1023: P 2:3(1) ack 2 win 4096
 6  0.539943 (0.0656)  bsdi.1023 > svr4.login: . ack 3 win 4096

 7  0.814582 (0.2746)  bsdi.1023 > svr4.login: P 2:3(1) ack 3 win 4096
 8  0.831108 (0.0165)  svr4.login > bsdi.1023: P 3:4(1) ack 3 win 4096
 9  0.940112 (0.1090)  bsdi.1023 > svr4.login: . ack 4 win 4096

10  1.191287 (0.2512)  bsdi.1023 > svr4.login: P 3:4(1) ack 4 win 4096
11  1.207701 (0.0164)  svr4.login > bsdi.1023: P 4:5(1) ack 4 win 4096
12  1.339994 (0.1323)  bsdi.1023 > svr4.login: . ack 5 win 4096

13  1.680646 (0.3407)  bsdi.1023 > svr4.login: P 4:5(1) ack 5 win 4096
14  1.697977 (0.0173)  svr4.login > bsdi.1023: P 5:7(2) ack 5 win 4096
15  1.739974 (0.0420)  bsdi.1023 > svr4.login: . ack 7 win 4096

16  1.799841 (0.0599)  svr4.login > bsdi.1023: P 7:37(30) ack 5 win 4096
17  1.940176 (0.1403)  bsdi.1023 > svr4.login: . ack 37 win 4096
18  1.944338 (0.0042)  svr4.login > bsdi.1023: P 37:44(7) ack 5 win 4096
19  2.140110 (0.1958)  bsdi.1023 > svr4.login: . ack 44 win 4096
```

Figure 19.2 TCP segments when `date` typed on Rlogin connection.

plus a CR/LF pair at the end. The next 7 bytes sent from the server to the client (line 18) are the client's prompt on the server host: `svr4 %` . Line 19 acknowledges these 7 bytes.

Notice how the TCP acknowledgments operate. Line 1 sends the data byte with the sequence number 0. Line 2 ACKs this by setting the acknowledgment sequence number to 1, the sequence number of the last successfully received byte plus one. (This is also called the sequence number of the next expected byte.) Line 2 also sends the data byte with a sequence number of 1 from the server to the client. This is ACKed by the client in line 3 by setting the acknowledged sequence number to 2.

19.3 Delayed Acknowledgments

There are some subtle points in Figure 19.2 dealing with timing that we'll cover in this section. Figure 19.3 shows the time line for the exchange in Figure 19.2. (We have deleted all the window advertisements from this time line, and have added a notation indicating what data is being transferred.)

We have labeled the seven ACKs sent from `bsdi` to `svr4` as *delayed ACKs*. Normally TCP does not send an ACK the instant it receives data. Instead, it delays the ACK, hoping to have data going in the same direction as the ACK, so the ACK can be sent along with the data. (This is sometimes called having the ACK *piggyback* with the data.) Most implementations use a 200-ms delay—that is, TCP will delay an ACK up to 200 ms to see if there is data to send with the ACK.

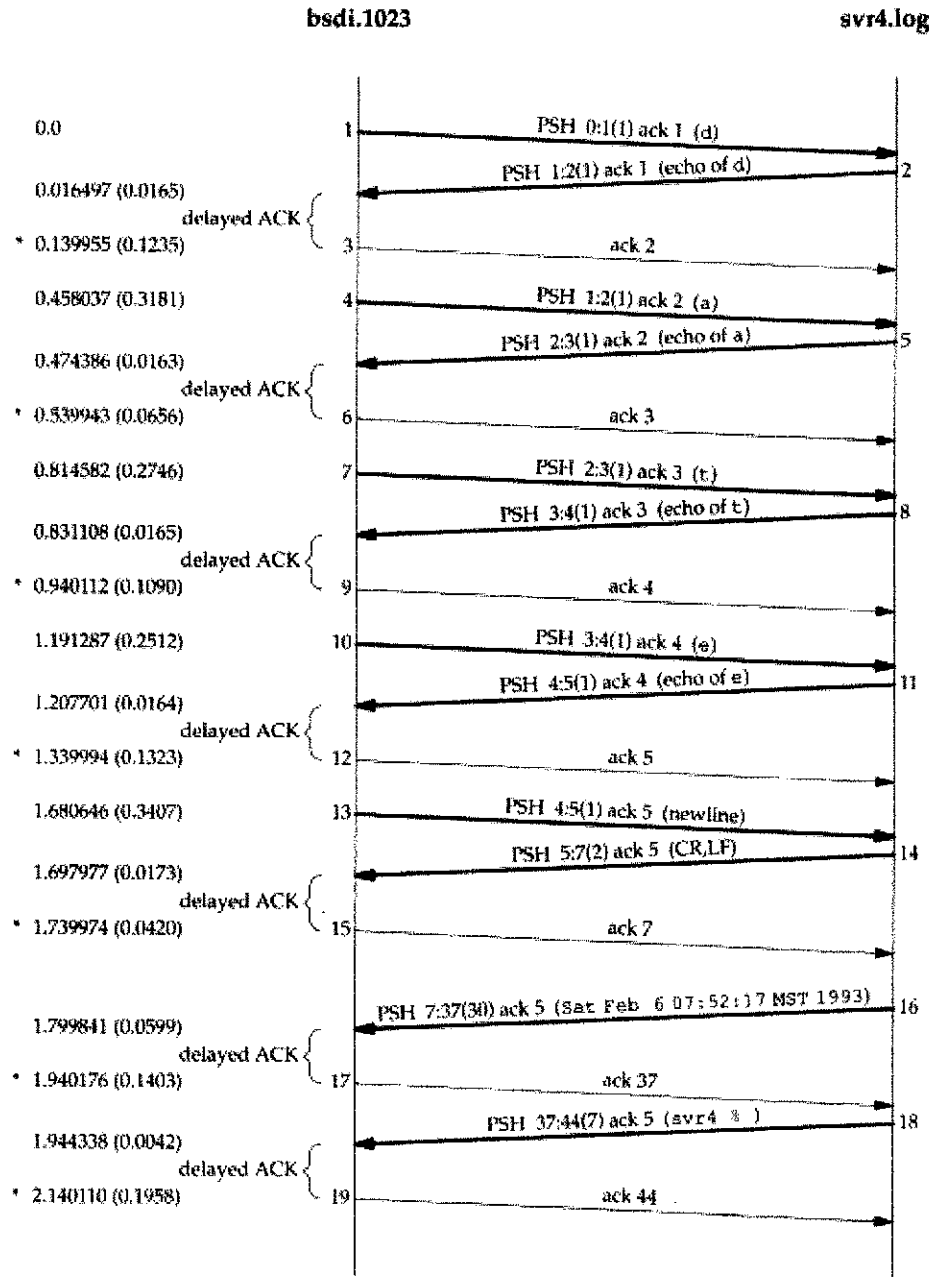

Figure 19.3 Time line of data flow for `date` command typed on an `rlogin` connection.

Rlogin

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 when a TCP connection has outstanding data that has not yet been acknowledged, small segments cannot be sent until the outstanding data is acknowledged. 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.)

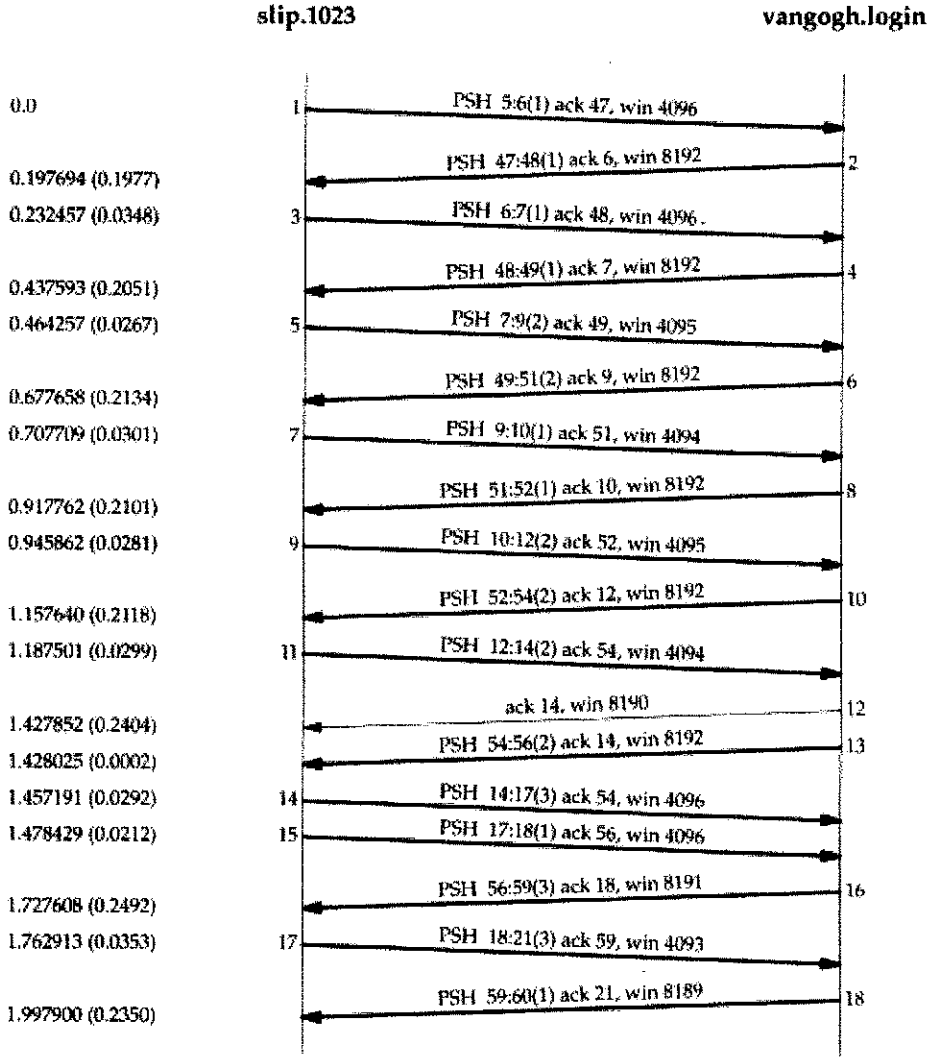

Figure 19.4 Data flow using `rlogin` between `slip` and `vangogh.cs.berkeley.edu`.

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.

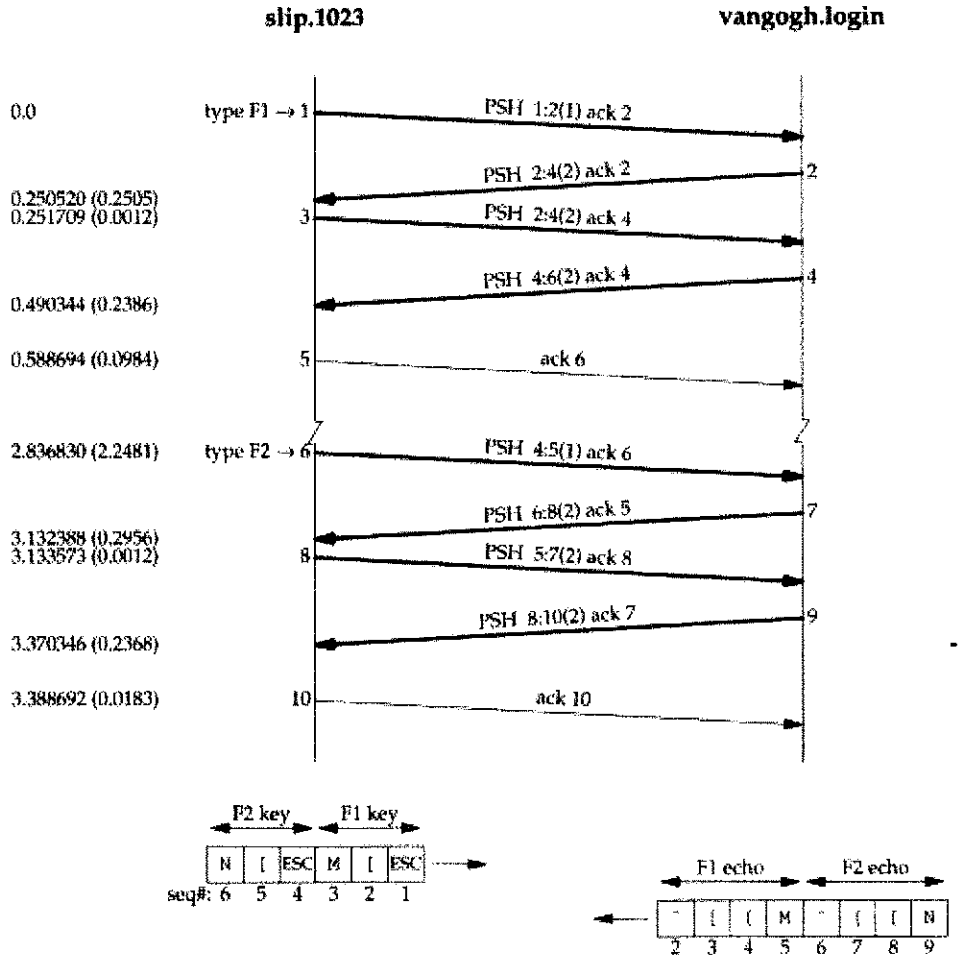

Figure 19.6 Time line for Figure 19.5 (watching the Nagle algorithm).

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. (See Exercise 19.3.)

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

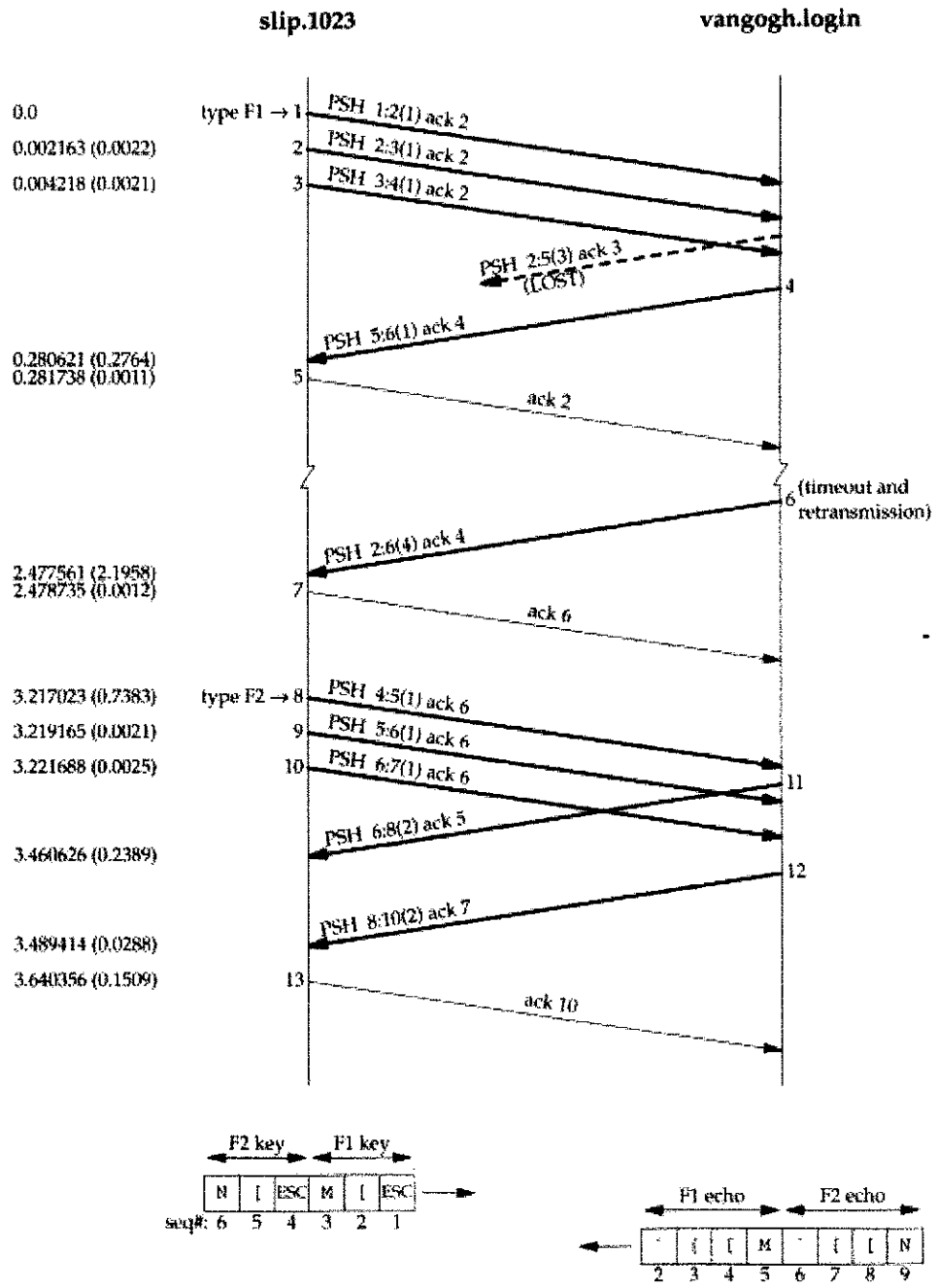

Figure 19.8 Time line for Figure 19.7 (Nagle algorithm disabled).

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).