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:

```
svr4 % 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 on the host bsdi 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   0.227682   ( 0.0077) svr4.5555 > bsdi.1027: . ack 5121 win 4096
10   0.233012   ( 0.0053) 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   ( 0.0043) 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 windows. 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.

ransmitting
ent can be
w that the
e and either
nowledged
9.4).

s, perhaps
ding small
ACKed and
rites (e.g.,
ly window

e algorithm
ding unac-
at "small"
n (b) only

ndow size
size of the
ease while

avoidance
with the

4 seconds
ive reads.

o stop the
we expect

(We have

he data at
uffer, and
ws every-

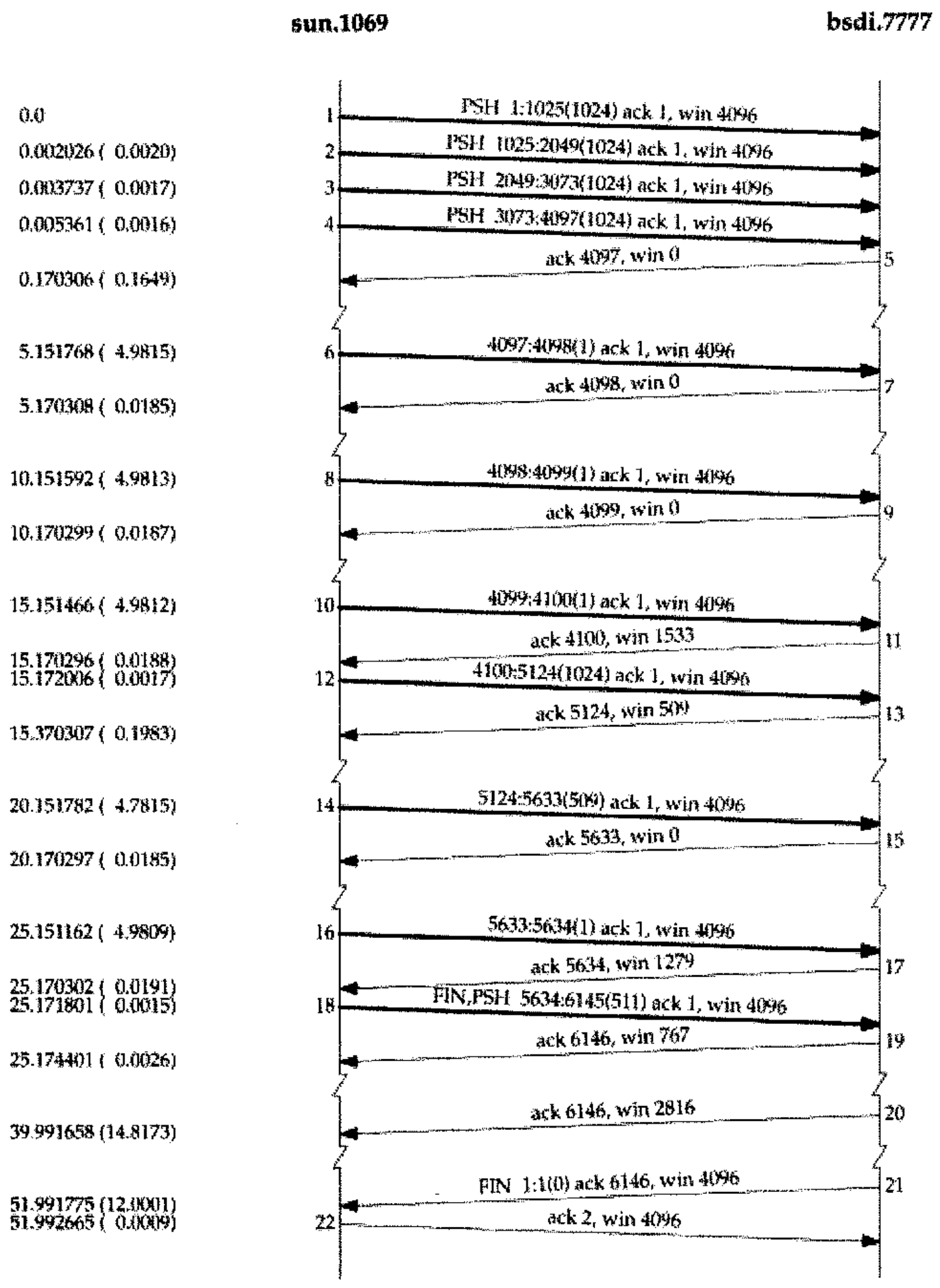

Figure 22.2 Time line showing receiver avoidance of silly window syndrome.

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 sending the data and receiving the ACK: 164.9, 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?