TCP Timeout And Retransmission

Chapter 21

TCP sets a timeout when it sends data and if data is not acknowledged before timeout expires it retransmits data.

* Timeout is based on round trip time measurement

Retransmission Used By TCP

Uses a doubling exponential back off

[Fig 21.1]

Lines 7-8 are retransmissions since disconnect ethernet cable. Timeout is doubled with upper limit of 64 seconds.

Round Trip Time (RTT) Measurement M

Originally : $R_{k+1} = \alpha R_k + (1- \alpha) M$ Recommended that $\alpha = 0.9$

Next retransmission timeout RTO $= \beta R_{current}$ Recommended that $\beta = 2$

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.

“ New ” Improved version

Error = M-A

$A = A + (0.125)_{g} * Error$

$D = D + (0.25)_{h} * (|error|-D)$

RTO = A + 4 D

Example But First An Aside:

Karns Algorithm - Problem: What is RTT when we receive an ACK for a segment that was retransmitted 1 or more times? Don’t know because we don’t know which segment the ACK is for.

Solution: Do not calculate a new retransmission timeout in this case (when ACK is for a retransmitted segment); use current timeout until an ACK for a segment without retransmissions is received.

RTT Example

This example starts with Figure 21.5 and continues in Figure 21.2. We show measurements in Fig. 21.3 and plot the measurements and the resulting calculations in Fig. 21.4 Lets look at how to Calculate the RTO timer value that is used in this real observation of a TCP session.

[Fig. 21.5] This is how the session started. We lost the first SYN segment and resent it after 6 seconds. Why did it resend after 6 seconds?

Calculations for the RTO for a SYN that is never ACKed :

A is always initialized to 0 seconds

D is always initialized to 3 seconds

In **first (and only first)** RTO calculation use 2D (instead 4D)

RTO = A + 2*D = 0 + 2 * 3 = 6 seconds is how long we will wait before retransmit the SYN

In this example, the first SYN is not acknowledged within the ≈ 6 seconds. Now use the

RTO = A + 4*D = 0 + 4 *3 = 12 times a doubling exponential back off (2^1) so = 24 seconds.

We now do get an ACK for the retransmission. So we are ready to move to Fig 21.2. But first an aside.

```
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.

Aside: If we had no ACK within 24 seconds: Use exponential back off factor of

(2^2) = 4 * RTO calculation = 48 seconds. However for this example we did get an ACK. It was for a duplicate sent datagram. Thus due to Karn's algorithm on duplicate datagrams, we do not update the RTO so it stays at the last value of 24 seconds.

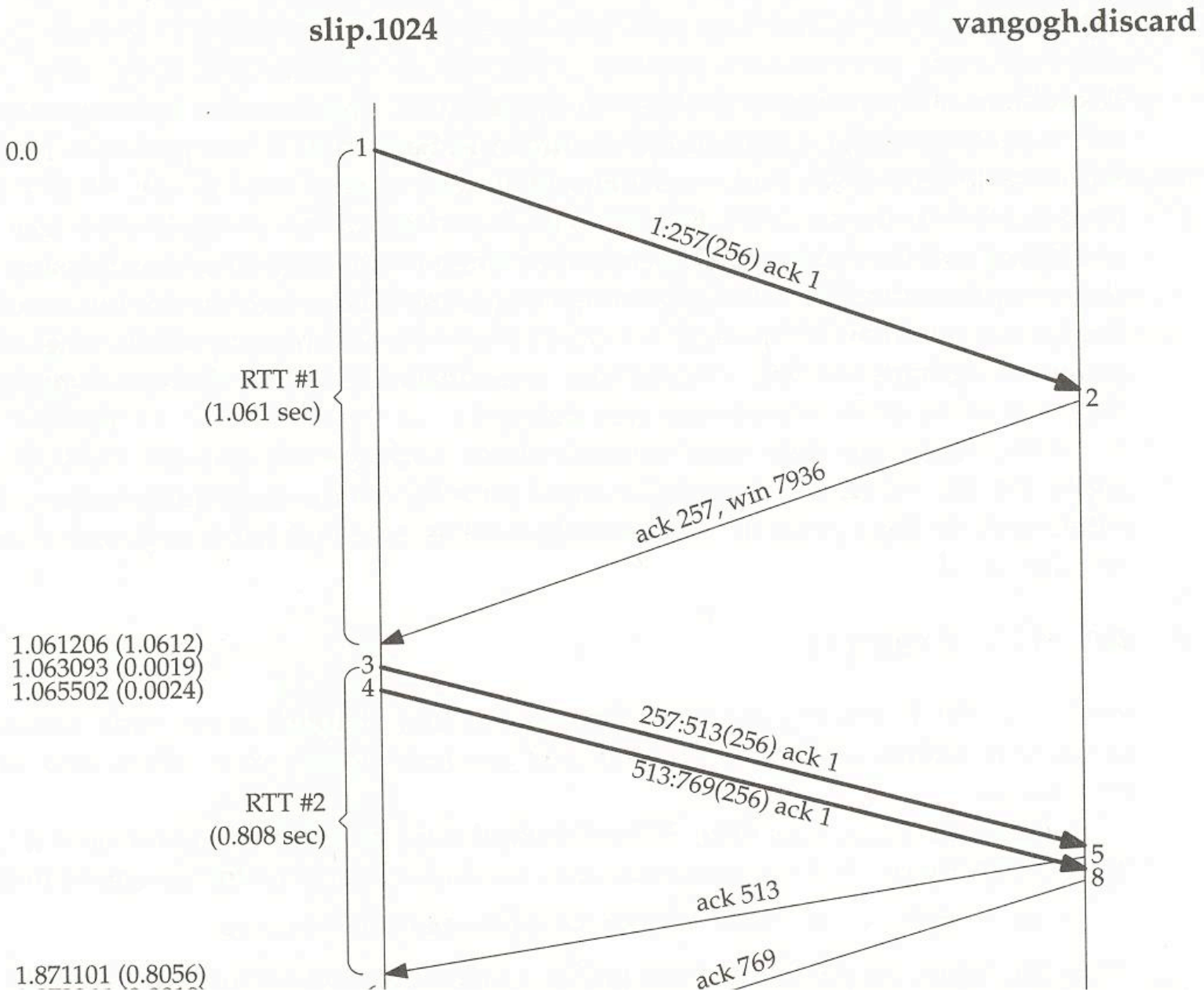
Now back to what happens next and is shown in Fig. 21.2

[Fig 21.2] Next 5 seconds shown in diagram instead of TCP dump output, removed window advertisements in Fig 21.2

[Fig 21.3] numbered segments in order transmitted or received on host.

Segment 1 to 3 :	3 ticks of a 500 ms TCP timer count only ticks
Segment 3 to 5 :	1 tick
Segment 6 to 11 :	2 ticks

Cannot use timer for two different simultaneous (overlapping) measurements


slip.1024
vangogh.discard
0.0
1
1:257(256) ack 1
RTT #1
(1.061 sec)
2
ack 257, win 7936
1.061206 (1.0612)
1.063093 (0.0019)
1.065502 (0.0024)
3
4
257:513(256) ack 1
513:769(256) ack 1
RTT #2
(0.808 sec)
5
8
ack 513
ack 769
1.871101 (0.8056)

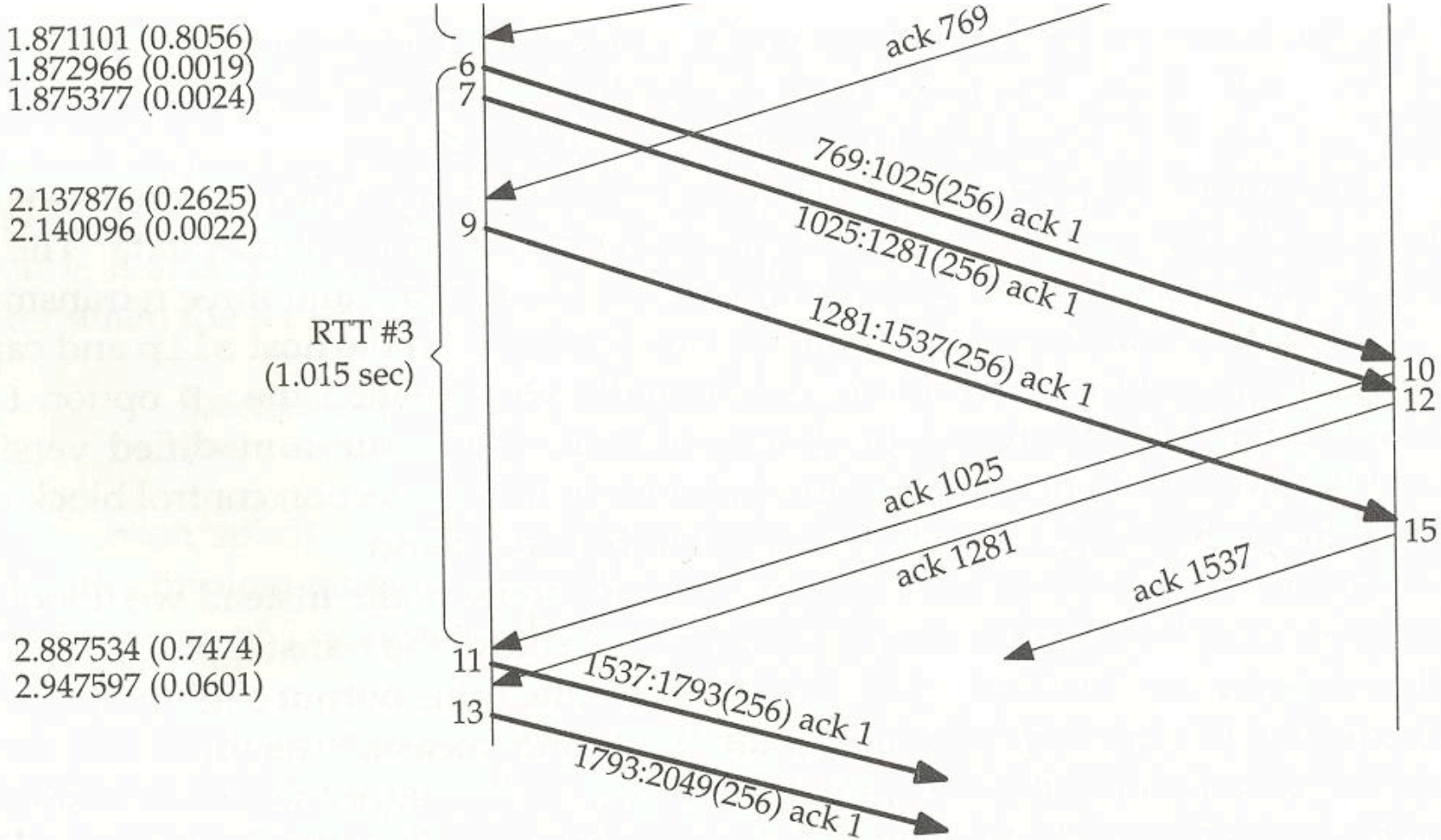


Figure 21.2 Packet exchange and RTT measurement.

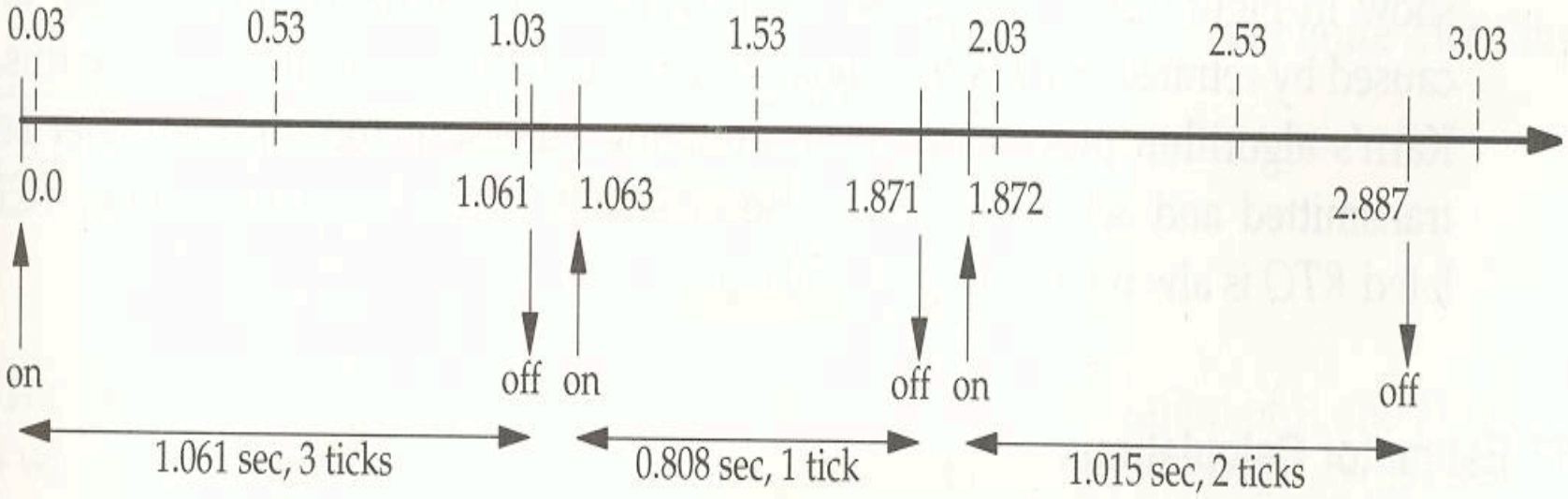


Figure 21.3 RTT measurements and clock ticks.

In our Fig 21.2 example, the RTO is 24 seconds when we send the first data segment.

Whenever we get the **first RTT measurement** we initialize the values of A and M again. This is an initialization for the first calculation using the first RTT measurement M. We initialize A and D to

A = M + 0.5
D= A/2
(This initialization again for the first used RTT measured point is not very clear in Stevens! Note A and D were initialized before the first transmission of the first SYN Segment, and now after receiving the first RTO measurement we are initializing A and D to something else!!!!!)

In our case we get our first ACK 3 clock ticks after the first data segment (Segment 1 in Fig 21.2) was sent.

Three clock ticks of a 500ms timer is 1.5 seconds, thus the 1.5 in the equation below

A = M + 0.5 = 1.5 + 0.5 = 2
D= A/2 = 2/2 = 1

We use RTO = A + 4*D = 2 + 4*1 = 6 seconds
Thus the timeout value in Fig 21.2 for segment #3 is 6 seconds. If we do not get an ACK in 6 seconds, we will assume lost and resend.

For the next ACK which is ACK 513 we measure 1 clock tick 500ms
A = 2 from above
D = 1 from above

Recall " New " Improved version equation for RTT timer calculation :

Error = M - A
$A = A + (0.125) * Error$ (g)
$D = D + (0.25) * (|error| - D)$ (h)
RTO = A + 4 D

For us M = one clock tick = 500 ms = 0.5 seconds
Error = M - A = 0.5 - 2 = - 1.5

A = A + 0.125 * Error = 2 - (0.125 * 1.5) = 1.8125
D = D + 0.25 (| Error | -D) = 1 + 0.25 (|-1.5 | - 1) = 1. 125
RTO = A + 4D = 1.8125 + 4* 1.125 = 6.32 seconds
Thus the timeout value in Fig 21.2 for segment #6 is 6 seconds. If we do not get an ACK in 6 seconds, we will assume lost and resend.

[Fig.21.4] Plot of these calculated values. Very first value of 24 seconds is not put in the figure.
We calculated 6 seconds and then 6.32 seconds, these are in the figure.
Actually use fixed-point not floating in implementations, floating point calculations used in book.

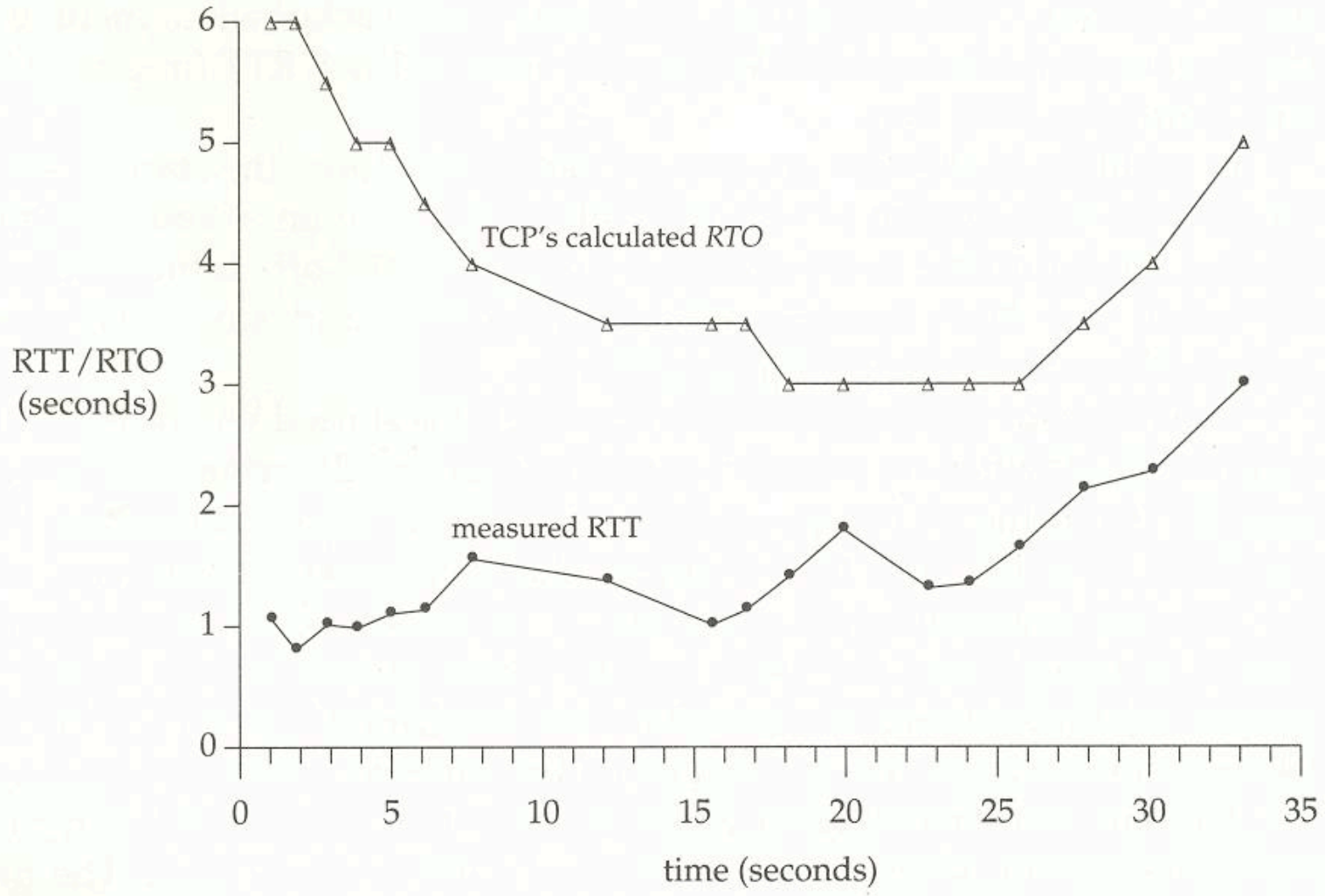
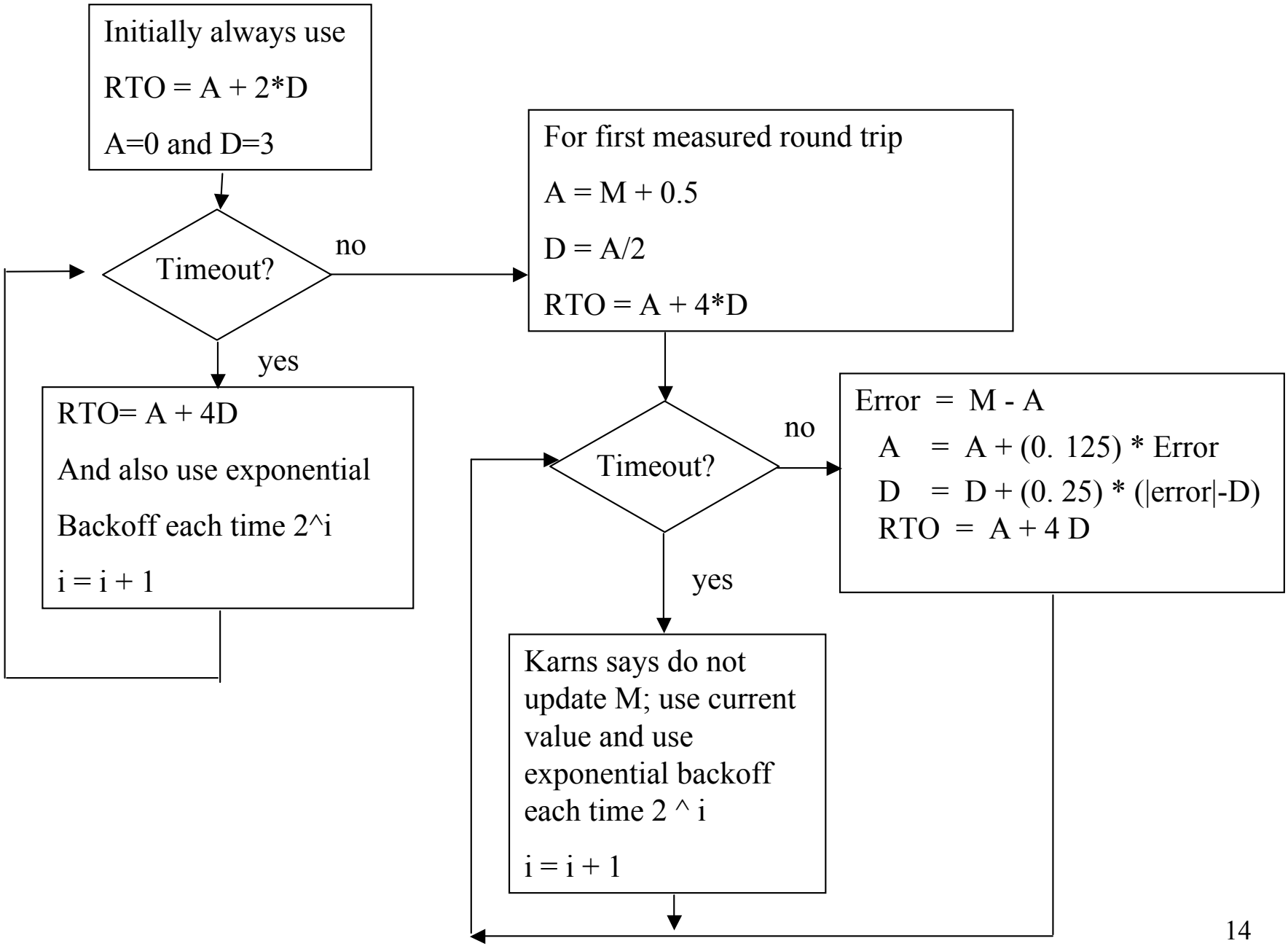


Figure 21.4 Measured RTT and TCP's calculated *RTO* for example.


Initially always use
RTO = A + 2*D
A=0 and D=3
Timeout?
no
yes
RTO= A + 4D
And also use exponential
Backoff each time 2^i
i = i + 1
For first measured round trip
A = M + 0.5
D = A/2
RTO = A + 4*D
Timeout?
no
yes
Error = M - A
A = A + (0. 125) * Error
D = D + (0. 25) * (|error|-D)
RTO = A + 4 D
Karns says do not update M; use current value and use exponential backoff each time 2 ^ i
i = i + 1

Congestion Example

[Fig 21.6]

Starting sequence number versus time

Retransmission is negative slope. Only 3 retransmissions occur. (one segment in each)

Look At The First Segment Loss At Time ≈ 10

[Fig 21.7] Zoom in at around 10 second point

This figure shows segments numbered according to their send or receive order with respect to host.

Removed unrelated segments 44,47, and 49; removed window advertisements slip=4096, vangogh=8192

Segment 45 gets lost and is not received.

ACK for up to 6657 is received in segment 58.

When vangogh receives segment 6913: 7169 (256) it repeats ACK for last received in sequence that is ACK 6657 again. This happens 9 times.

On slip receipt of the third **duplicate** ACK for the same thing: a retransmission of the first (and in this case only) missing segment is done.

* Note in this case only the single segment that was not received is retransmitted. After retransmission of segment 63 which is 6657:6913 (256) sender keeps on going with segment 67 8961:9217 (256), segment 69 and segment 71.

Continued….

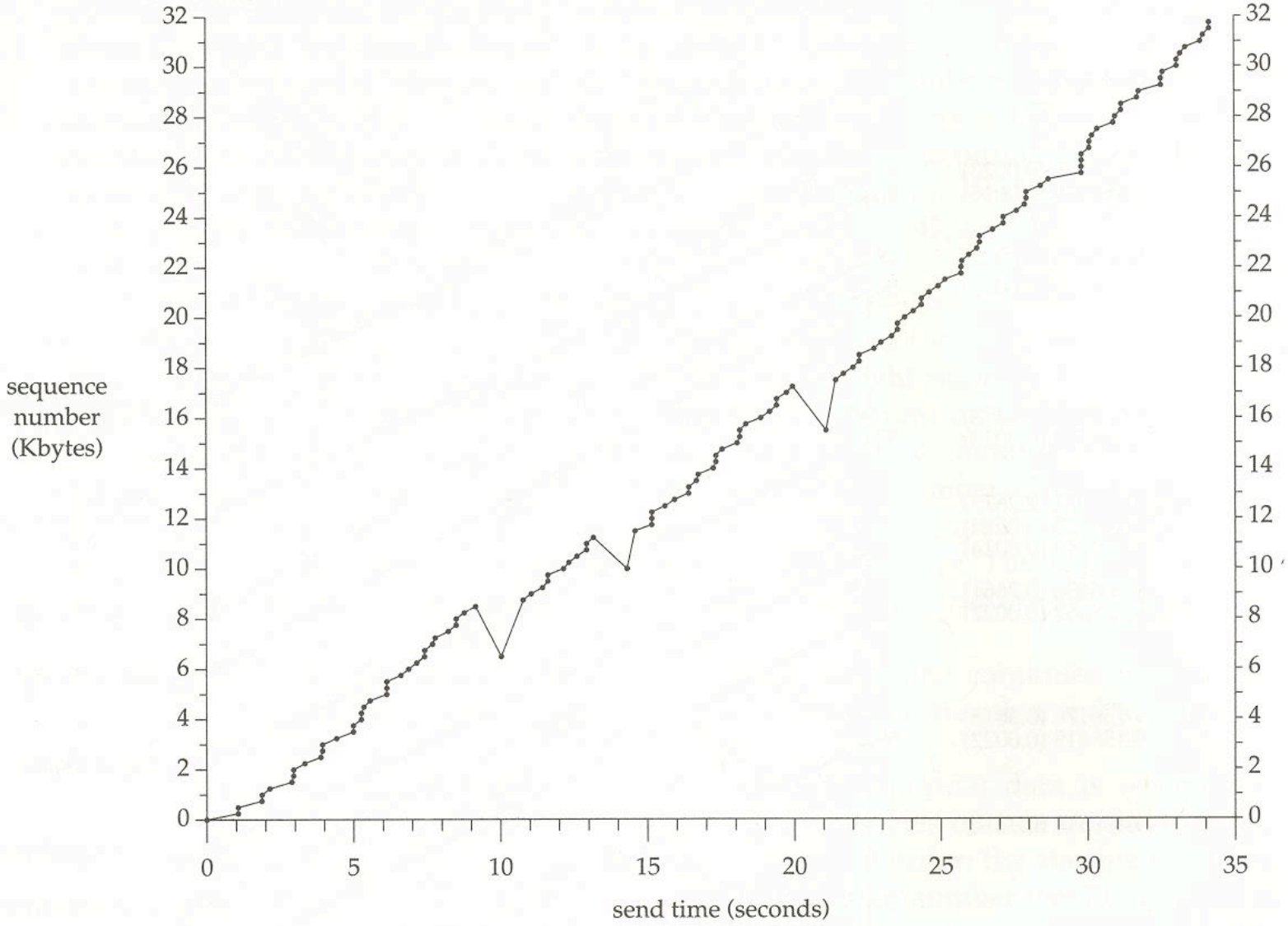


Figure 21.6 Sending of 32768 bytes of data from `slip` to `vangogh`.

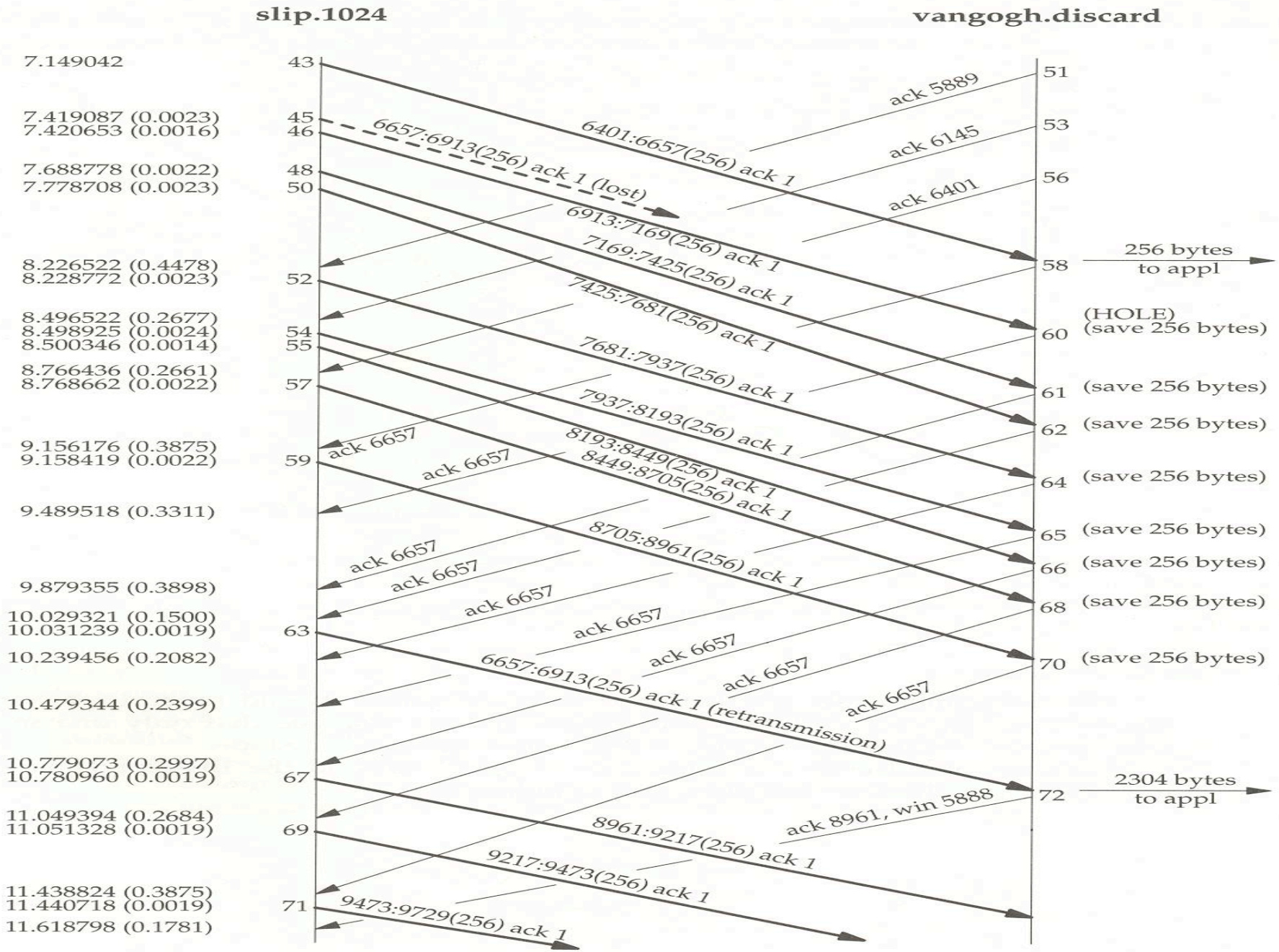


Figure 21.7 Packet exchange for retransmission around the 10-second mark.

* TCP does not wait for the other end to acknowledge the retransmission.

  - At the receiver (vangogh) normal data is received in sequence (segment 43)

  - 256 bytes of data is passed up to the user process.

  - Next received segment (segment 46) is out of order, it starts at 6913 but 6657 is expected.

  - TCP saves out of order and immediately ACK's with highest sequence number received in order plus 1 (6657)

  - Next seven segments received by vangogh are also out of order but are saved. Duplicate ACK of 6657 sent for each.

  - When missing data segment 63 6657: 6913 (256) arrives receiving TCP already has 6657 through 8960 so an ACK for 8960 + 1 is sent.

  - Window of 5888 which is 8192 - 2304 is advertised since receiving process has not yet read the buffer.

So how do we decide how big the sliding window (cwnd) in the sender should be?

Congestion Avoidance Algorithm

Two possible indications of Packet Loss

- Timeout occurring
- Receipt of duplicate ACK's

If congestion, we want to slow down transmission.

Congestion avoidance and slow start work together.

Congestion window cwnd
Slow start threshold size ssthresh

Algorithms:

1. cwnd = 1 segment
   ssthresh = 65535 Bytes
   } Initially

2. TCP never sends more than minimum of [cwnd and receiver's advertised window].
3. When congestion encountered set ssthresh = minimum of ($^1/_2$) (cwnd, receivers advertise window) or at least 2 segments. {Rounded down to a segment size multiple}

   And if congestion seen by timeout set
   cwnd = one segment

4. For each ACK of new data (does not happen if a duplicate ACK)
   if cwnd ≤ ssthresh use **slow start** which means increment cwnd by one segment for each ACK

   else increment cwnd by (This is not slow start so is congestion avoidance)

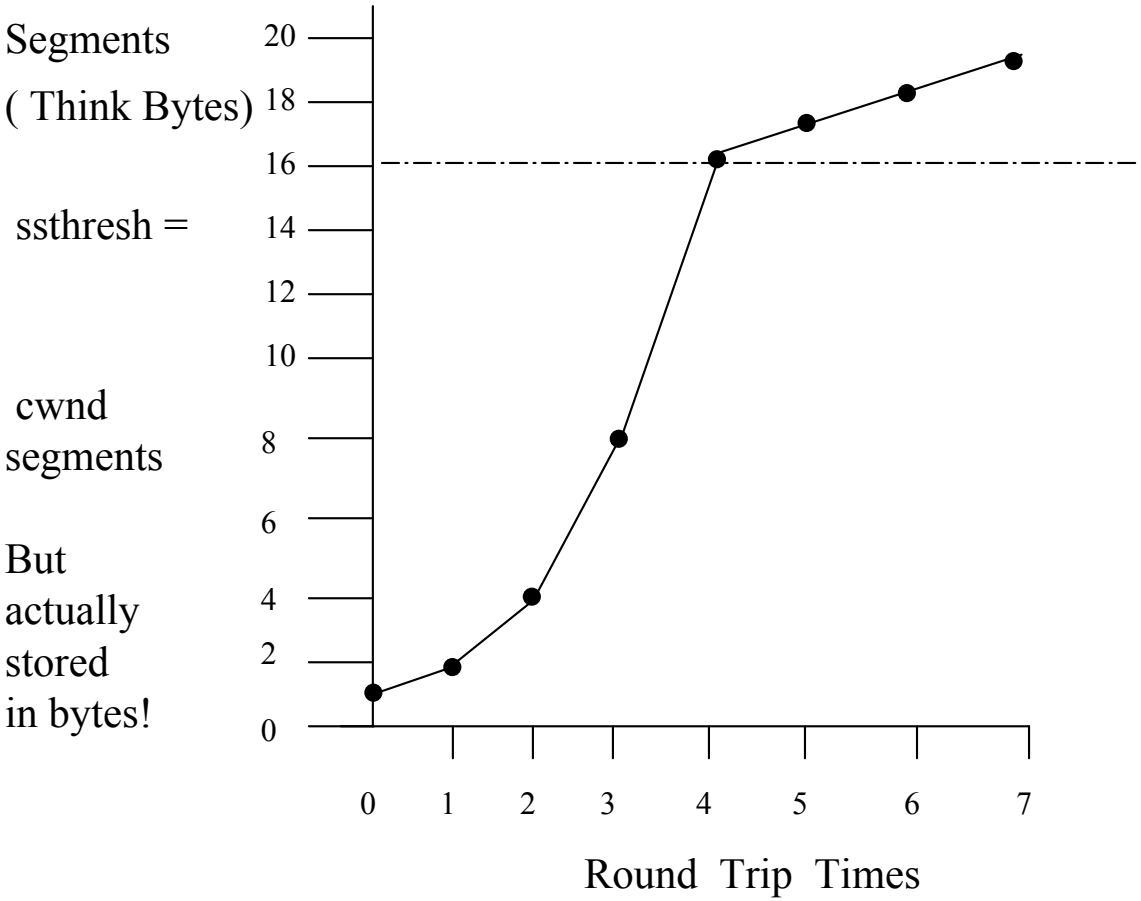
$$\frac{\text{segsize} * \text{segsize}}{\text{cwnd}} + \frac{\text{segsize}}{8}$$

   For each ACK received

Note: Maximum increment in congestion avoidance allowed is 1 MSS segment.

Example : Assume congestion previously occurred when cwnd was 32 segments; as a result of seeing congestion when cwnd = 32 then ssthresh = 16 is our starting condition

This is Fig 21.8 redrawn:



- Send one segment at time 0
- If ACK received send 2 segments at time 1 (cwnd =2)
- If two ACK's received send 4 segments at time 2 (cwnd = 4)
- At time 3 send 8 segments (cwnd =8)
- At time 4 send 16 segments (cwnd = 16)

-At time 5 gets incremented to above ssthresh so no longer in slow start region
thus instead (in congestion avoidance region)

increment cwnd by $\frac{\text{segsize x segsize}}{\text{cwnd}} + \frac{\text{segsize}}{8}$

Limit this increase to one segment maximum

NOTE:
Stevens says $\frac{\text{segsize}}{8}$ term **should not** be included but it is in some code. We will always use it also.
(See RFC 2001 top page 4 for comment on old implementations incorrectly adding this term).

NOTE:
Congestion avoidance is flow control imposed by the sender based on network congestion
Advertised window is flow control imposed by receiver based on available buffer space.

Example in a minute……

Fast Retransmit and Fast Recovery Algorithms

TCP is required to generate an immediate acknowledgement (a duplicate ACK) when an out of order segment is received.

It is sent with “No Delay”.

We wait for 3 or more received duplicate ACKS in a row to make sure its not just a temporary reordering. Note that 4 ACKs for the same segment is considered as an ACK and 3 duplicate ACKS = 4 total!

Then perform retransmission of what appears to be the missing segment without waiting for retransmission timer to expire

Called **fast retransmit** algorithm.

Next congestion avoidance not slow start is performed (well sort of, one slow start type increment and then Congestion avoidance, see example later to see what this means. RFC 2851 page 4 says “when cwnd and ssthresh are equal the sender may use either slow start or congestion avoidance”).

Called **fast recovery** algorithm.

Since data must still be flowing through network if generating duplicate ACK’s do not go to slow start.

Fast Recovery Algorithm :

1. When 3rd duplicate ACK received set ssthresh to (½)(cwnd) rounded down to a segment size multiple.
2. Retransmit missing segment
3. Set cwnd to ssthresh plus (3 x segment size).
4. For each duplicate ACK received **after the retransmitted segment**, increment cwnd by segment size and transmit another TCP segment if allowed by new value of cwnd.
5. For **first new** ACK, set cwnd to ssthresh (**ssthresh still has the** value in step 1) This should be ACK of retransmission from step **2**. This ACK should acknowledge all intermediate segments sent between lost TCP segment and receipt of duplicate ACK.

This is congestion avoidance since slowing down to one - half rate when packet lost.

Whew….need an example!

Simple Example Of Slow Start

Given mss = 256 bytes
cwnd = 256 bytes
ssthresh = 65535 bytes

For each ACK received
cwnd = cwnd + 1 segment
cwnd = cwnd + 256
= 512, 768, 1024, 1280, etc

Assuming congestion does not occur, continue until advertised window (receiver buffer!) limits what sender can transmit

Congestion Example (Back to our chapter's long example from several figures)

Assume advertised window = 8192 on vangogh
[Fig 21.9]

Initial SYN is lost (know by timeout) so retransmit
cwnd = 256 (set cwnd = one segment we are slow start)
=> ssthresh = min of ½ (cwnd, advertised window)
or at least 2 segments
use 2 segments = 2 x 256 = 512 bytes

=> cwnd = one segment = 256

Segment# (Figure 21.2)	Action			Variable	
	Send	Receive	Comment	*cwnd*	*ssthresh*
	SYN		initialize	256	65535
	SYN		timeout retransmit	256	512
	ACK	SYN, 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.

Second SYN is acknowledged

Next we have first data acknowledged ACK 257 and we have cwnd $\leq$ ssthresh
Since 256<= 512. Note test is performed before put new value on the line in Fig. 21.9

$\Rightarrow$ cwnd = cwnd + 1 segment = 512 bytes

We now have second data segment acknowledged ACK 513and still cwnd $\leq$ ssthresh

$\Rightarrow$ cwnd = cwnd + 1 segment = 768 bytes

Now we have

cwnd > ssthresh

No longer in slow start

Third data segment acknowledged (ACK 769)

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

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

cwnd = 885

Fourth Data Segment Acknowledged

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

cwnd = 991

[Fig 21.10] is plot of these results

Look at places where congestion occurred and we enter fast retransmit.

[Fig 21.11] and [Fig 21.7] congestion shown

After 3 duplicate ACKs

1. ssthresh = (½ * cwnd) rounded down to a segment size multiple

   = ½ * 2426 = 1,213 round down to 4 x 256 = 1024

2. cwnd = ssthresh + 3 x segment size

   = 1024 + 3 x 256

   = 1792

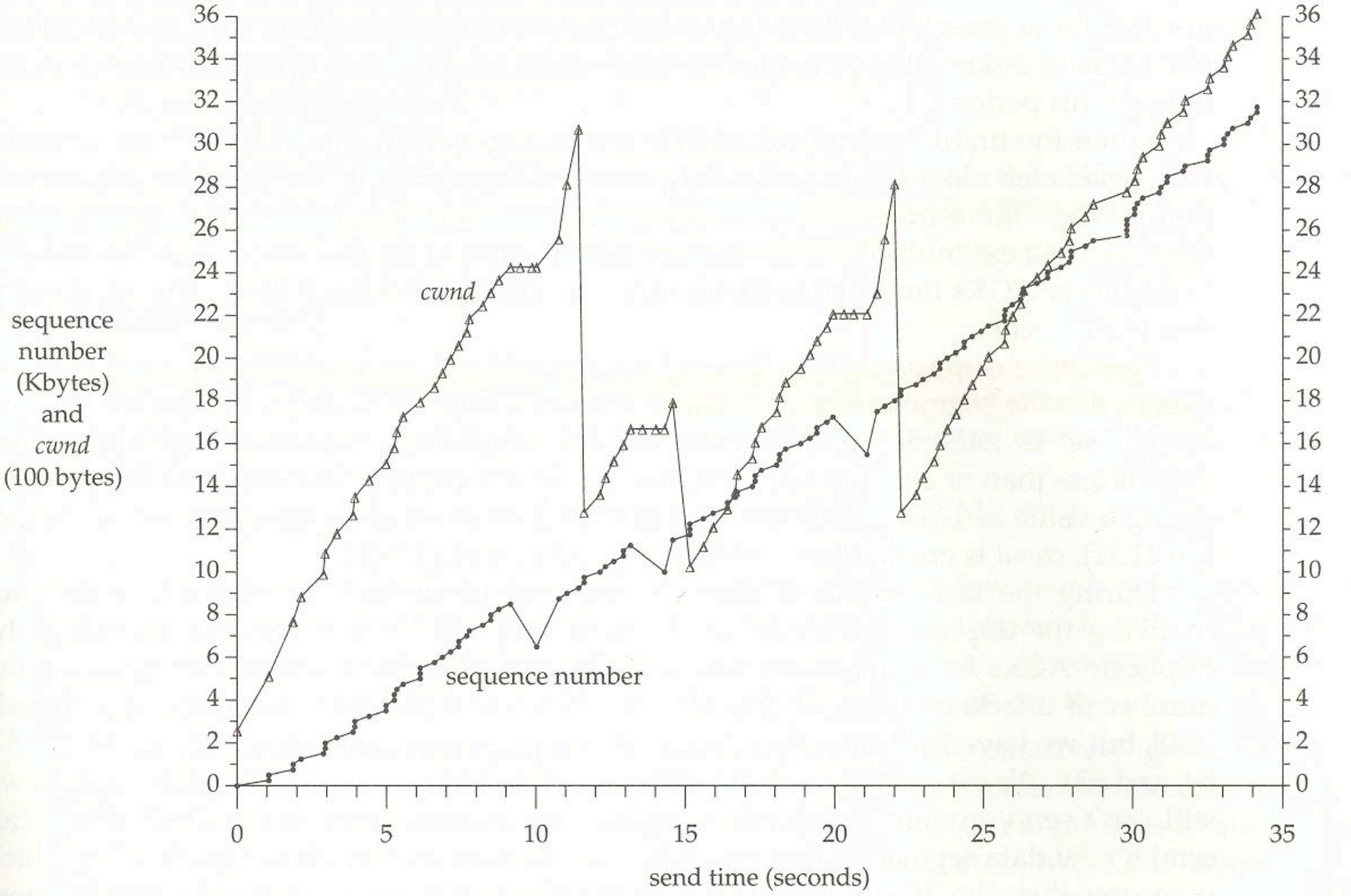


Figure 21.10 Value of *cwnd* and send sequence number while data is being transmitted.

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).

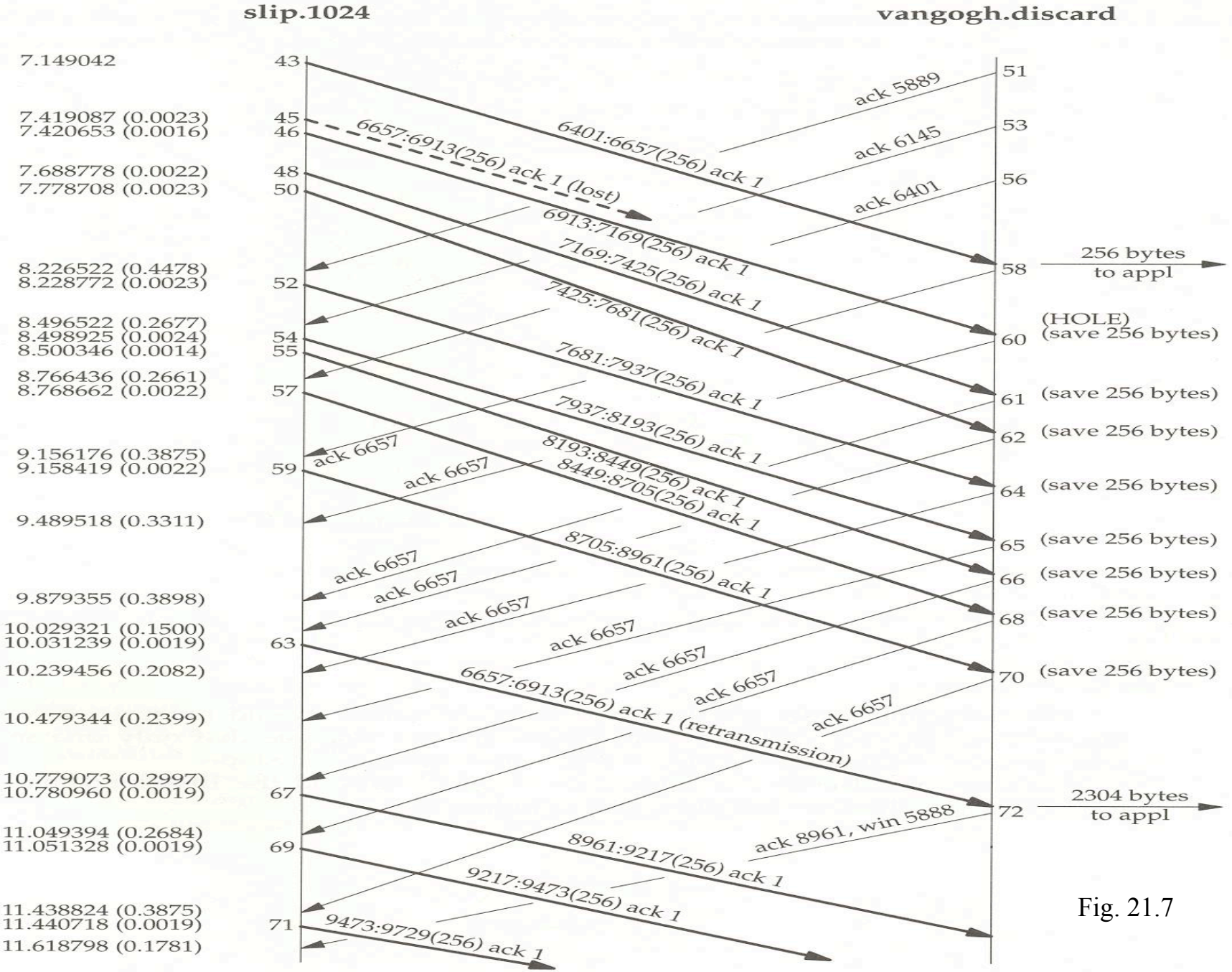


Fig. 21.7

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

The Retransmission Is Now Sent

5 more duplicate ACK's arrive, increment cwnd by segment size each time and transmit another TCP segment if allowed by value of cwnd

cwnd = cwnd + 256

Segment 64	=	1792 + 256 = 2048
Segment 65	=	2048 + 256 = 2304
Segment 66	=	2304 + 256 = 2560
Segment 68	=	2560 + 256 = 2816
Segment 70	=	2816 + 256 = 3072

Now the new data ACK arrives, set cwnd to ssthresh = 1024
"Normal algorithm" takes over which means we must do the test

is cwnd $\leq$ ssthresh ? **since this is true we are in a slow start part**
1024 $\leq$ 1024 **of the "normal algorithm"**

This means we must increment cwnd by one segment for each ACK

cwnd = cwnd + 256
= 1024 + 256
= 1280

Not Shown In [Fig 21.11]

Next New ACK

Since cwnd $\leq$ ssthresh is not true

$$1280 \leq 1024$$

Increment cwnd by

$$\frac{\text{segsize x segsize}}{\text{cwnd}} + \frac{256}{8}$$

$$\text{cwnd} = \text{cwnd} + \frac{256 \text{ x } 256}{1280} + \frac{256}{8} = 1363$$

Comment on step 4 in fast recovery algorithm:

4. For each duplicate ACK received **after the retransmitted segment**,

   increment cwnd by segment size and transmit another TCP segment

   if allowed by new value of cwnd.

[Fig 21.7 and Fig 21.11]

After segment 64 **received** in Fig 21.7 cwnd = 2048 (see slide 31)

but we have 9 x 256 =2304 unacknowledged bytes (segments 45, 46, 48, 50, 52, 54, 55, 57, 59).

Since can only send the minimum of cwnd (2048) and advertised window (which is 8192 in this problem), can not send any more data.

Must wait until cwnd grows. cwnd grows upon receipt of every duplicate ack.

Segment 65 makes cwnd 2304, so we still cannot send a new segment.

Segment 66 makes cwnd 2560 , segment 67 is sent.

PERSIST: THE SECOND TCP TIMER

What happens if a non-zero window update is lost? How do we avoid deadlock?

[Fig 20.3] Window Update Review

Sender uses a persist time to cause sender to periodically probe to find out if window has increased => window probes

Example:

[Fig 22.1]

Segment 13 advertises a window of 0, causing sender to stop

After persist timer expires, send the next single byte (TCP is allowed to send 1 byte of data beyond end of a closed window)

ACK comes back with a window of 0 causing another wait

Persist timer uses TCP exponential back off but bounds of 5 to 60 seconds used if calculate values below 5 or above 60

Unlike retransmission timeout which has a total timeout value of 2 minutes (even though page 298-299 had 9 minutes), the persist probes never give up, thus 60 second intervals forever

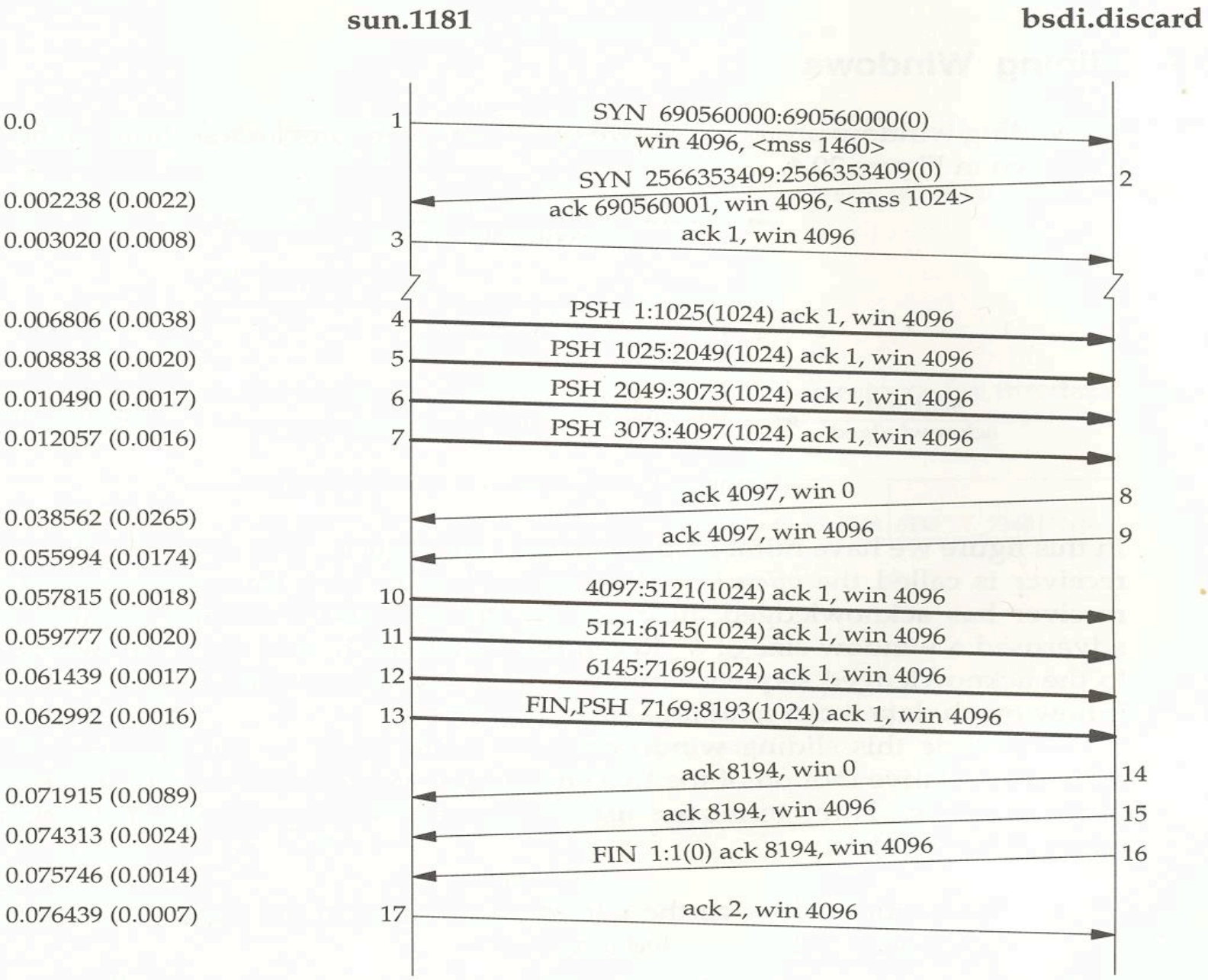


Figure 20.3 Sending 8192 bytes from a fast sender to a slow receiver.

```
 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.

SILLY WINDOW SYNDROME

If this happens (Silly Window Syndrome) small amounts of data flow instead of full size MSS

Receiver can cause by advertising small windows instead of waiting until a larger window

Sender can cause by not waiting to send more data

To prevent:

1. Receiver does not increase advertised window unless one full segment size (MSS) or one-half the receiver's buffer space, which ever is smaller.

2. Sender does not transmit unless one of:

    a. Full size MSS can be sent

    b. We can send at least one-half of the maximum sized window that the other end has advertised

    c. We can send everything we have and:

        1. We have no outstanding unacknowledged data OR

        2. Nagle is Disabled

KEEPALIVE: TCP TIMER NUMBER THREE

No data flows across an idle TCP connection

Optional timer, not part of TCP specification but found in most implementations

- Not really needed since “connection” defined by end points
- “connection” is up as long as destination and source are alive

Typically used to detect half open connections and de-allocate server resources in for example an FTP session

Default value is two hours, after 2 hours server sends a probe to client, if no response after 10 probes at 75 seconds apart, server assumes client no longer there and closes connection.

2MSL TIMER: TCP TIMER NUMBER FOUR

Measures time a connection has been in the TIME_WAIT state

MSL is defined as 2 minutes but is implementation specific

2MSL is twice the two minutes

200 ms: TCP TIMER NUMBER FIVE

RECALL 200 ms: TCP Timer, lets call that TCP timer number five.