

## Chapter 20 – TCP Bulk Data Flow

TCP implements a byte-oriented sliding window protocol.

The two objectives of the protocol are:

Ensure reliable delivery

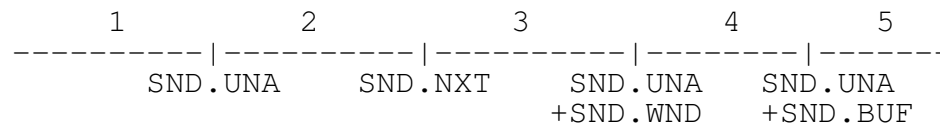
Provide pacing

Don't allow a fast sender to overwhelm a slow receiver

Don't allow a fast (or slow!) sender to congest the network.

### TCP State Variables:

#### Send Sequence Space



1 – old sequence numbers which have been acknowledged

2 – sequence numbers of unacknowledged data

3 – sequence numbers allowed for new data transmission

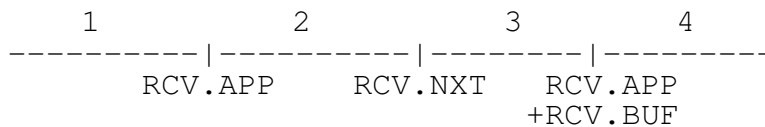
4 – future sequence numbers can't be seen but may be queued

5 – sender app is put to sleep when sequence number of buffered data reaches here

#### Send Sequence Variables

SND.UNA	– send unacknowledged = value of last <i>ack</i> received
SND.NXT	– send next = next sequence number to be transmitted
SND.WND	– send window = value of last <i>window</i> received
SND.BUF	– maximum amount of transmit buffer space available to this connection
SND.UP	– send urgent pointer
SND.WL1	– segment sequence number used for last window update
SND.WL2	– segment acknowledgment number used for last window update
ISS	– initial send sequence number

## Receive Sequence Space



- 1 – old sequence numbers which have been consumed by app
- 2 – old sequence numbers which have been acked but not consumed
- 3 – sequence numbers allowed for new reception
- 4 – future sequence numbers which are not yet allowed

## Receive Sequence Variables

RCV.APP	– next byte to be consumed by the receiving app
RCV.NXT	– receive next = sequence number of next expected byte of data
RCV.WND	– offered window = $RCV.APP + RCV.BUF - RCV.NXT$
RCV.UP	– receive urgent pointer
RCV.BUF	– maximum amount of recv buffer space available to connection
IRS	– initial receive sequence number

Examples:

Equal speed sender and receiver in figure 20.1 shows:

- How acks are typically (but not always) generated after two segments are received.
- Segment received w/ ack timer running => generate ACK / turn off timer
- Reduction of offered window when application hasn't received data
- Ack at 8 generated by the delayed ACK timer.
- Where does the ack at 10 come from??

Slow receiver in figure 20.3 shows:

- That acks are not always generated after two segments are received.
- It is legal to offer a window of 0 => sender must desist from sending
- If you do you must later issue a window update (what if lost?)

Conclusion: Your mileage may vary... any two exchanges of exactly the same data may differ significantly in dynamics.

## Window management

Offered window –

Value of "window" advertised by receiver  
Actually computed as  $rcv.app + rcv.buf - rcv.nxt$   
[ $rcv.nxt, rcv.nxt + window$ ]

Usable window – Offered window minus whatever sent.  
[ $snd.nxt, snd.una + window$ ]

Window movement terminology:

close – Left edge moves right  
open – Right edge moves right  
shrink – Right edge moves left... Discouraged but must be supported.

## Push flag

Original intent

Sender to tell receiving TCP to allow any blocking read to complete.

Problem

No way in the API to tell sender TCP to set the flag.

Solution

Always set it.

## Urgent data pointer

Not truly out of band data

Used by telnet and rlogin

Points to the last byte of urgent data (RFC) or the next byte beyond (implementations).

6. Suppose receive buffer space is 12,000 bytes, `rcv.app` = 2000 and `rcv.nxt` is 8000.
- a. What values will be sent in outgoing packets for  
WINDOW =  
ACK =
  - b. Suppose a packet with `SEQ` = 9000 and `length` = 1000 is now received.  
What will outgoing window and ack be now?  
WINDOW =  
ACK =
  - c. Suppose a packet with `SEQ` = 8000 and `length` = 1000 is now received  
What will outgoing window and ack be now?  
WINDOW =  
ACK =
  - d. Finally assume that the application reads 5000 bytes.  
What will outgoing window and ack be now?  
WINDOW =  
ACK =
7. Suppose transmit buffer space is 20,000, `snd.ntx` = 28,000  
`snd.una` = 22,000 and a packet is received with `ACK` = 26000  
and `WINDOW` = 22000. After the incoming packet is processed,
- a. Assuming no unsent bytes are presently buffered, how many  
bytes can the sending application buffer before it is  
forced to sleep?
  - b. Suppose that the full complement of bytes are presently buffered,  
how many bytes may the sending TCP send before it  
must stop sending.

## Window size determination

Objective for *link layer* protocols:

Find the *minimum* size window that will allow the sender to transmit continuously without blocking.

The ack for the first byte transmitted should come back *just* before the sender runs out of window space.

For a fixed bandwidth (e.g. STDM) circuit this size is simply the bandwidth delay product.

$W.S = \min \text{ roundtrip propagation delay in seconds} * \text{transmit rate in bits / second.}$

Some examples:

T1 with a 60 ms RTT =  $1.544 / 8 * 10^6 * 60 * 10^{-3} = 11,580$  bytes

T3 with a 60 ms RTT =  $45.0 / 8 * 10^6 * 60 * 10^{-3} = 337,500$  bytes

Objective for *transport layer* protocols:

Find the *minimum* size window that will allow the sender to transmit as possible without

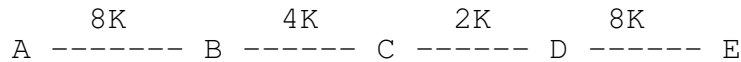
- overwhelming the receiver (easy to implement)
- overly congesting the internet (hard to implement)

For *transport layer protocols* using the bandwidth delay measure is *not* inherently stable..

If the congestion caused by your session causes the RTT to increase, and you increase the window size so you won't have to block you will cause more congestion which will cause your RTT to increase causing you to increase the window size....

### Effective Throughput (Bandwidth) determination:

Heterogenous links complicate the problem since effective bandwidth is not likely to be your output link bandwidth.



Suppose host A send 1Kb packets at a rate of 8 pkt/sec

Packets pile up at B at a rate of 4 pkt/sec  
Packets pile up at C at a rate of 2 pkt/sec  
Packets are delivered to E at a rate of 2 pkt/sec

How can host determine the maximum rate at which packets should be sent??

--> By the rate at which ACKs are returned.

Here since 2 packets / second actually arrive at E, 2 acks per second will be generated.  
Thus 2 acks/second will be delivered to A in the steady state.

Thus, a possible choice for window size in the for a real world transport might be approximate bandwidth delay product:

Window size = effective throughput \* average latency

Little's Law tells us that this approach is equivalent to driving the "system" at 100% utilization.

However, this analysis ignores the roles of congestion and the stochastic nature of interarrival and (effective) service times.. ("Our" packets's service times depend upon the stochastic nature of the "competition".)

In an M/M/1 queue throughput =  $\min(\lambda, \mu)$  but response time becomes infinite as  $\lambda \rightarrow \mu$ .

## Optimizing throughput while minimizing congestion and response in the "real world"

No known algorithmic method exists... Instead a collection of heuristics have been developed.

Original TCP was completely driven by the *offered window* limit on the number of bytes outstanding in the network.

Send until blocked by offered window size

If packets get dropped, lack of ack will eventually lead to timeout

Timeout will result in a big blast of a full window of retransmissions

Heuristics use the following idea:

Start with a small window

Gradually increase it

When trouble occurs shrink it back...

TCP Tahoe (late 1980's, Van Jacobson) modifications included *slow start* and *congestion avoidance*.

## Slow start

Motivation: The bottleneck link problem just discussed.

In original TCP ...

- Sender injects a full offered window of segments as FAST as it can.
- Segments queue at bottleneck routers.
- Lack of buffer space leads to massive packet drops
- Massive packet drops lead to time outs and massive retransmissions
- Inevitable result is bad performance
- Possible result is congestion collapse.

Objective:

Limit the rate at which packets are injected to the rate acks can be returned.

Implementation:

- Add a new state variable called *cwnd* (congestion avoidance window)
- Sender can only send the  $\min(\text{usable\_window}, \text{congestion\_avoidance\_window})$ .
- Initial value of *cwnd* is 1 MSS.

Value of *cwnd* is increased by one each time a segment is acked

	<i>cwnd</i>
Send	1
wait (1 ack recvd)	
Send	2
Send	2
wait (2 acks rcvd)	
Send	4
Send	4
Send	4
Send	4

Value quickly climbs up to the offered window ... or a drop occurs..

## Bottom line:

NO TCP session will *ever* have more than the "offered window" in the pipe  
But sometime even that is too much.  
*Forcing a TCP session into slow start will reduce congestion.*  
This is the basis for the Random Early Drop heuristic.