EEE 545

Computer Networks 1

6. ABSTRACT SLIDING WINDOWS

6. Abstract Sliding Windows

▶ In this chapter we take a general look at how to build reliable data-transport layers on top of unreliable lower layers. This is achieved through a retransmit-on-timeout policy; that is, if a packet is transmitted and there is no acknowledgment received during the timeout interval then the packet is resent.

▶ In addition to reliability, we also want to keep as many packets in transit as the network can support. The strategy used to achieve this is known as sliding windows.

6.1 Building Reliable Transport: Stop-and-Wait

▶ Retransmit-on-timeout generally requires sequence numbering for the packets, though if a network path is guaranteed not to reorder packets then it is safe to allow the sequence numbers to wrap around surprisingly quickly

▶ We will assume conventional numbering. Data[N] will be the Nth data packet, acknowledged by ACK[N].

▶ In the stop-and-wait version of retransmit-on-timeout, the sender sends only one outstanding packet at a time. If there is no response, the packet may be retransmitted, but the sender does not send Data[N+1] until it has received ACK[N]. Of course, the receiving side will not send ACK[N] until it has received Data[N]; each side has only one packet in play at a time.

# 6.1 Building Reliable Transport: Stop-and-Wait

Stop and Wait
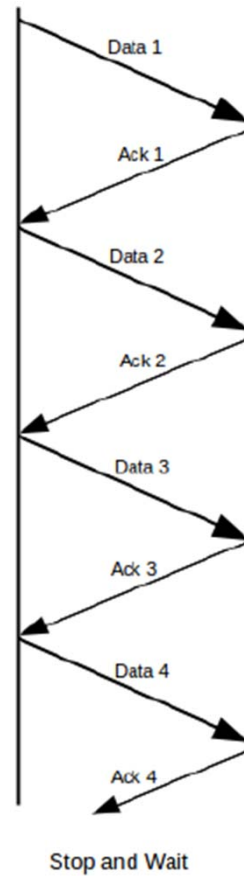
6.1.1 Packet Loss

▶ The diagram below illustrates a lost Data packet, where the sender is the host sending Data and the Receiver is the host sending ACKs. The receiver is not aware of the loss; it sees Data[N] as simply slow to arrive.

Sender   Receiver

Data[N]

Timeout   Data[N]

ACK[N]

Lost Data
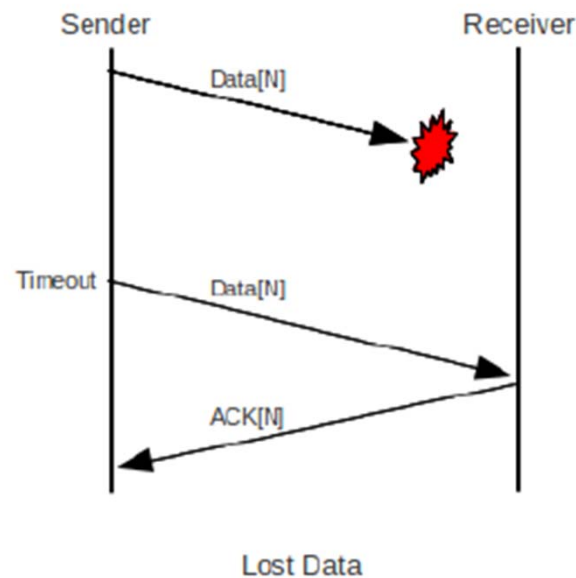
6.1.1 Packet Loss

▶ The diagram below, by comparison, illustrates the case of a **lost** ACK. The receiver has received a duplicate Data[N]. We have assumed here that the receiver has implemented a **retransmit-on-duplicate** strategy, and so its response upon receipt of the duplicate Data[N] is to retransmit ACK[N].


Lost ACK
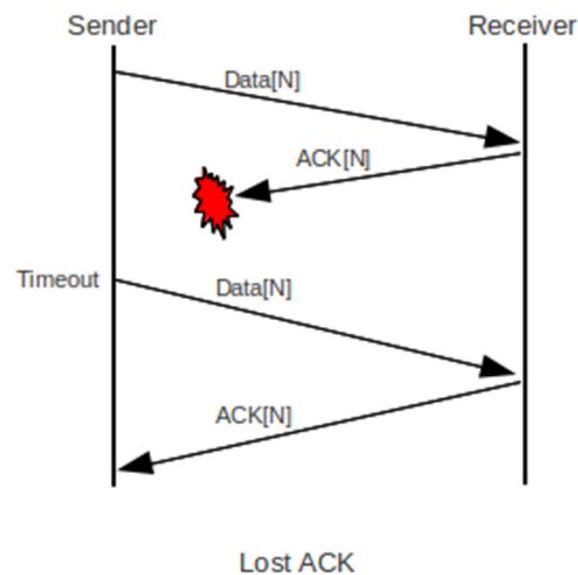
6.1.1 Packet Loss

▶ As a final example, note that it is possible for ACK[N] to have been delayed (or, similarly, for the first Data[N] to have been delayed) longer than the timeout interval. Not every packet that times out is actually lost!

▶ In this case we see that, after sending Data[N], receiving a delayed ACK[N] (rather than the expected ACK[N+1]) must be considered a normal event.

▶ In principle, either side can implement **retransmit-on-timeout** if nothing is received. Either side can also implement **retransmit-on-duplicate**; this was done by the receiver in the second example above but not by the sender in the third example (the sender received a second ACK[N] but did not retransmit Data[N+1]).

▶ At least one side must implement **retransmit-on-timeout**; otherwise a lost packet leads to deadlock as the sender and the receiver both wait forever. The other side must implement at least one of retransmit-on duplicate or retransmit-on-timeout; usually the former alone. If both sides implement retransmit-on-timeout with different timeout values, generally the protocol will still work.
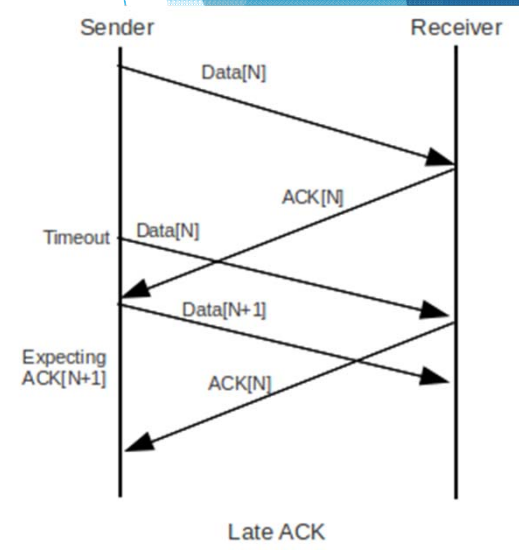
6.1.1 Sorcerer's Apprentice Bug

▶ A strange thing happens if one side implements **retransmit-on-timeout** but both sides implement **retransmition-on-duplicate**.

▶ Let us imagine that an implementation uses this strategy, and that the initial ACK[3] is delayed until after Data[3] is retransmitted on timeout. In the following diagram, the only packet retransmitted due to timeout is the second Data[3]; all the other duplications are due to the retransmit-on-duplicate strategy.

▶ All packets are sent twice from Data[3] on. The transfer completes normally, but takes double the normal bandwidth.
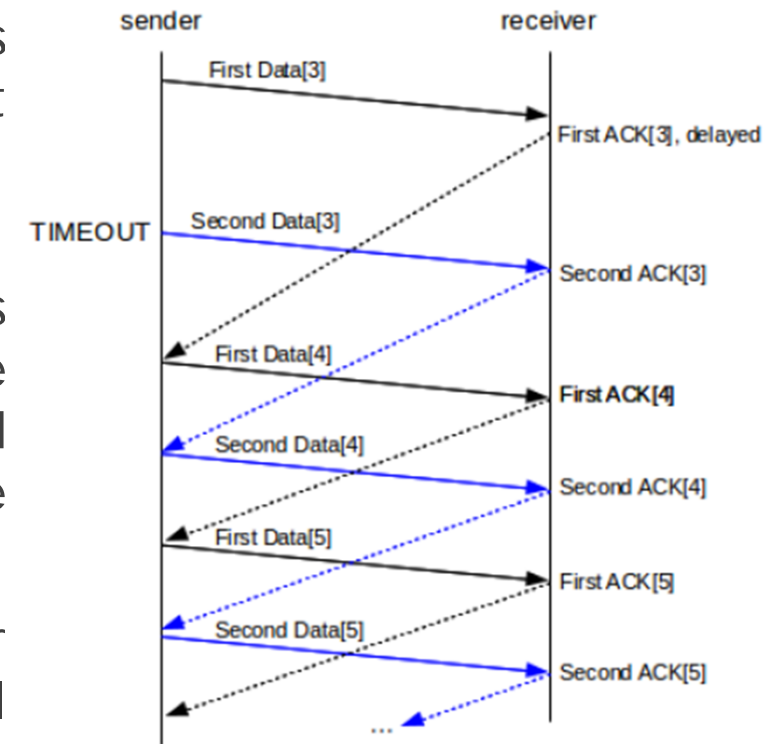
The Sorcerer's Apprentice bug
First transmissions are in black
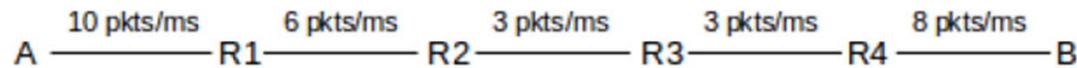Second transmissions are in blue

6.1.3 Flow Control

▶ Stop-and-wait also provides a simple form of flow control to prevent data from arriving at the receiver faster than it can be handled. Assuming the time needed to process a received packet is less than one RTT, (round-trip time, or RTT: the time between sending a packet and receiving a response.) the stop-and-wait mechanism will prevent data from arriving too fast. If the processing time is slightly larger than RTT, all the receiver has to do is to wait to send ACK[N] until Data[N] has not only arrived but also been processed, and the receiver is ready for Data[N+1].

▶ For modest per-packet processing delays this works quite well, but if the processing delays are long it introduces a new problem: Data[N] may time out and be retransmitted even though it has successfully been received; the receiver cannot send an ACK until it has finished processing. One approach is to have two kinds of ACKs: ACKWAIT[N] meaning that Data[N] has arrived but the receiver is not yet ready for Data[N+1], and ACK_{GO}[N] meaning that the sender may now send Data[N+1]. The receiver will send ACK_{WAIT}[N] when Data[N] arrives, and ACK_{GO}[N] when it is done processing it.

6.2. Sliding Windows

▶ We can improve overall throughput by allowing the sender to continue to transmit, sending Data[N+1] (and beyond) without waiting for ACK[N]. We cannot, however, allow the sender get too far ahead of the returning ACKs. Packets sent too fast, as we shall see, simply end up waiting in queues, or, worse, dropped from queues. If the links of the network have sufficient bandwidth, packets may also be dropped at the receiving end.

▶ Now that, say, Data[3] and Data[4] may be simultaneously in transit, we have to revisit what ACK[4] means: does it mean that the receiver has received only Data[4], or does it mean both Data[3] and Data[4] have arrived? We will assume the latter, that is, ACKs are cumulative: ACK[N] cannot be sent until Data[K] has arrived for all K≤N.

6.3. Linear Bottlenecks

▶ The minimum bandwidth, or path bandwidth, is 3 packets/ms. (For the figure)

```
        10 pkts/ms        6 pkts/ms        3 pkts/ms        3 pkts/ms        8 pkts/ms
  A ——————————— R1——————————— R2——————————— R3——————————— R4 ——————————— B
```

▶ The slow links are R2–R3 and R3–R4. We will refer to the slowest link as the bottleneck link; if there are (as here) ties for the slowest link, then the first such link is the bottleneck. The bottleneck link is where the queue will form. If traffic is sent at a rate of 4 packets/ms from A to B, it will pile up in an ever-increasing queue at R2. Traffic will not pile up at R3; it arrives at R3 at the same rate by which it departs.

▶ Furthermore, if sliding windows is used (rather than a fixed-rate sender), traffic will eventually not queue up at any router other than R2: data cannot reach B faster than the 3 packets/ms rate, and so B will not return ACKs faster than this rate, and so A will eventually not send data faster than this rate. At this 3 packets/ms rate, traffic will not pile up at R1 (or R3 or R4).

6.3.1 Simple fixed-window-size analysis

▶ We will analyze the effect of window size on overall throughput and on RTT(Round-Time-Trip). Consider the following network path, with bandwidths now labeled in packets/second.

```
        infinitely fast      1 pkt/sec      1 pkt/sec      1 pkt/sec      1 pkt/sec
A ─────────────────── R1 ──────────── R2 ──────────── R3 ──────────── R4 ──────────── B
```

▶ We will assume that in the backward B→A direction, all connections are infinitely fast, meaning zero delay; this is often a good approximation because ACK packets are what travel in that direction and they are negligibly small. In the A→B direction, we will assume that the A→R1 link is infinitely fast, but the other four each have a bandwidth of 1 packet/second (and no propagation-delay component).

▶ This makes the R1→R2 link the bottleneck link; any queue will now form at R1. The "path bandwidth" is 1 packet/second, and the RTT is 4 seconds.

6.3.1.1 Case 1: winsize = 2

Time	A	R1	R1	R2	R3	R4	B
T	sends	queues	sends	sends	sends	sends	ACKs
0	1,2	2	1				
1			2	1			
2				2	1		
3					2	1	
4	3		3			2	1
5	4		4	3			2
6				4	3		
7					4	3	
8	5		5			4	3
9	6		6	5			4

RTT= 8-4=4s

▶ Note the brief pile-up at R1 (the bottleneck link!) on startup. However, in the steady state, there is no queuing. Real sliding-windows protocols generally have some way of minimizing this "initial pileup".

▶ RTT=4sec, Throughput=2 packets in 4 second=0.5packet/sec

6.3.1.2 Case 2: winsize = 4

T	A sends	R1 queues	R1 sends	R2 sends	R3 sends	R4 sends	B ACKs
0	1,2,3,4	2,3,4	1				
1		3,4	2	1			
2		4	3	2	1		
3			4	3	2	1	
4	5		5	4	3	2	1
5	6		6	5	4	3	2
6	7		7	6	5	4	3
7	8		8	7	6	5	4
8	9		9	8	7	6	5

RTT= 8-4=4s

▶ Note the brief pile-up at R1 (the bottleneck link!) on startup. However, in the steady state, there is no queuing. Real sliding-windows protocols generally have some way of minimizing this "initial pileup".

▶ RTT=4sec, Throughput=4 packets in 4 second=1packet/sec

6.3.1.2 Case 3: winsize = 6

T	A sends	R1 queues	R1 sends	R2 sends	R3 sends	R4 sends	B ACKs
0	1,2,3,4,5,6	2,3,4,5,6	1				
1		3,4,5,6	2	1			
2		4,5,6	3	2	1		
3		5,6	4	3	2	1	
4	7	6,7	5	4	3	2	1
5	8	7,8	6	5	4	3	2
6	9	8,9	7	6	5	4	3
7	10	9,10	8	7	6	5	4
8	11	10,11	9	8	7	6	5
9	12	11,12	10	9	8	7	6
10	13	12,13	11	10	9	8	7

RTT= 10-4=6s

▶ Note the brief pile-up at R1 (the bottleneck link!) on startup. However, in the steady state, there is no queuing. Real sliding-windows protocols generally have some way of minimizing this "initial pileup".

▶ RTT=6sec, Throughput=6 packets in 6 second=1packet/sec