TCP Strategies

Slow Start

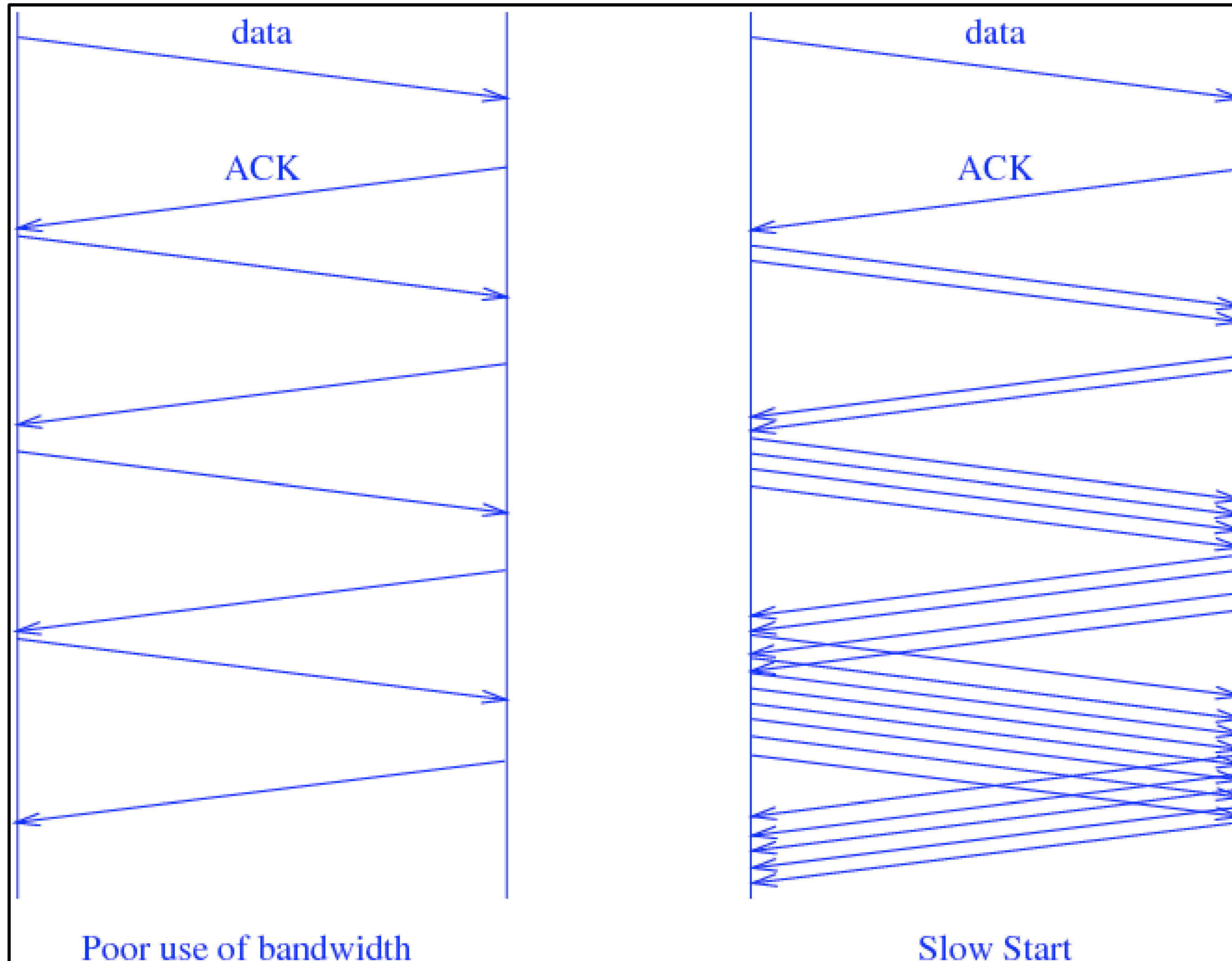
- *Slow start* tries to measure the path congestion by measuring the number of ACKs coming back
- It uses a *congestion window*, which is this estimate
- This is another constraint on sending additional to the advertised window

TCP Strategies

Slow Start

- The window is initialised to the maximum segment size of the destination
- A variable, the *threshold* is initialised to 64KB
- Every time a timely ACK is received, the congestion window is increased by one segment

TCP Strategies



- so initially, we can send one segment
- then 2 at a time
- then 4 ...

TCP Strategies

Slow Start

- This is actually an exponential increase in the congestion window over time
- It is “slow” in comparison with an earlier version of TCP that started by blasting out segments as fast as possible even before the performance of the network was known
- The doubling continues until we reach the current threshold (which started at 64KB)

TCP Strategies

Congestion Avoidance

- At this point we change to the *congestion avoidance* phase
- Now we increase the congestion window by

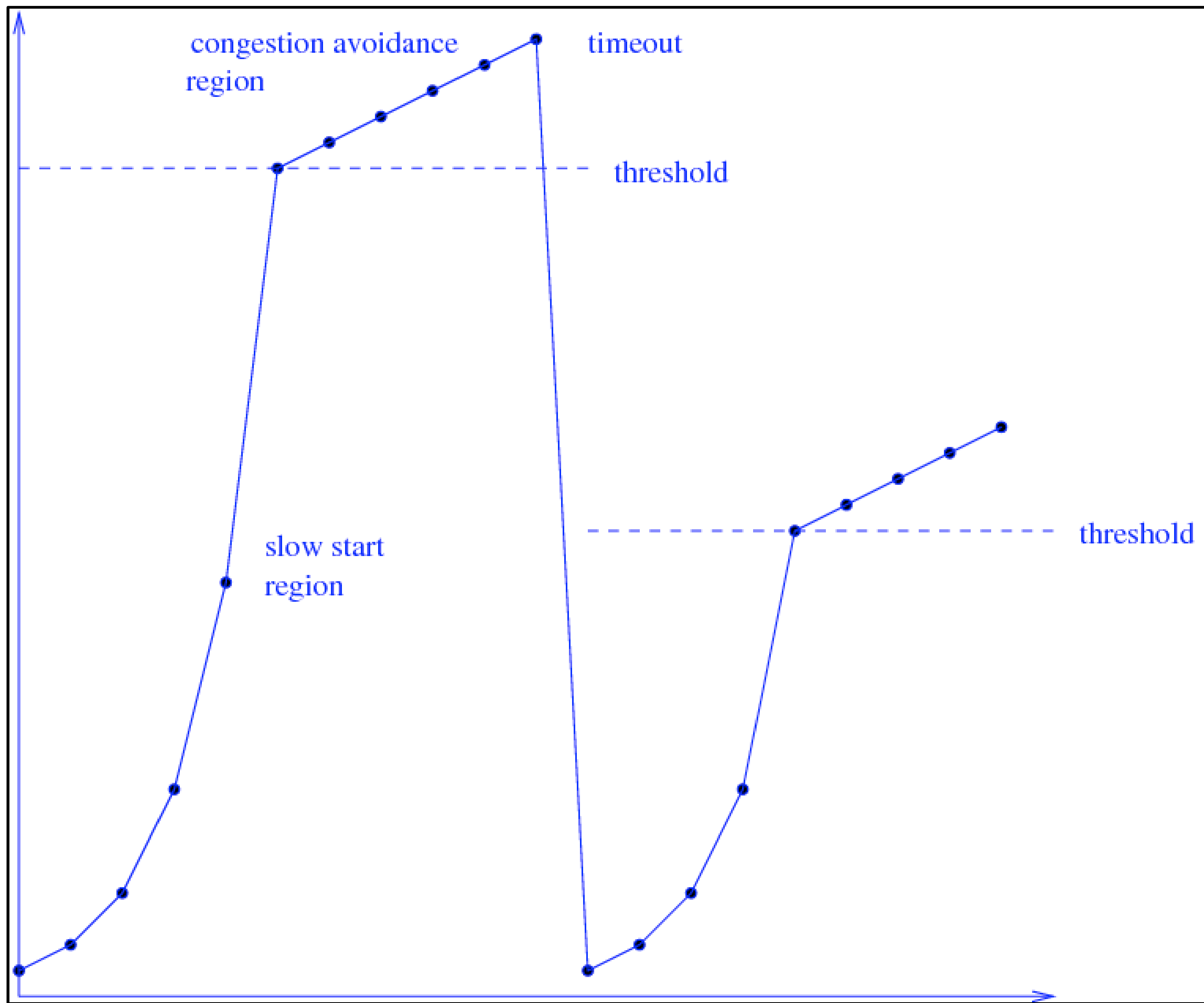
1/(congestion window size)

for each timely ACK

TCP Strategies

Congestion Avoidance

- Eventually the network's limit will be reached and a congested router somewhere will start dropping segments
- The sender will see this when ACKs stop coming
- At this point, the threshold is set to half the current congestion window (there's a mistake in the textbook!), the current window drops back to one segment and we go back to slow start



- The sender eventually converges on a rate that is neither too fast, nor too slow

TCP Strategies

Slow Start and Congestion Avoidance

- If conditions on the network change, it will soon adapt to the new rate, be it faster or slower
- If there is no congestion on the network, the rate increases until it reaches the advertised window: the limiting factor is then the destination, not the network

TCP Strategies

Fast Retransmit and Fast Recovery

- When an out-of-order segment is received the TCP protocol calls for an immediate ACK: it must not be delayed
- This is to inform the sender as soon as possible that something is wrong
- Jacobson's *Fast Retransmit* strategy takes the idea that several duplicated ACKs is indeed indicative of a lost segment

TCP Strategies

Fast Retransmit and Fast Recovery

- The argument is that one or two duplicated ACKs might simply be due to out-of-order delivery (recall IP is unreliable)
- Three or more means something is wrong
- If this happens, retransmit the indicated segment immediately

TCP Strategies

Fast Retransmit and Fast Recovery

- Next, do congestion avoidance and do not go into slow start: this is the *fast recovery* strategy
- We don't want slow start as the duplicate ACKs indicate that later data have reached the destination and is buffered there
- So data is still flowing and we don't want to abruptly cut the flow by doing slow start

TCP Strategies

Explicit Congestion Notification

- A newer congestion mechanism is ECN
- This lets routers put a mark on packets as they pass to indicate the route is becoming congested
- This can be used by the hosts as prior notification to slow down before packets start being dropped

TCP Strategies

Explicit Congestion Notification

- ECN uses bits 6 and 7 in the TOS field of the IPv4 header: these were previously unused and set to 0
- Bit 6 is "ECN capable transport" (ECT), to indicate that the sender is aware of the ECT mechanism: this provides backwards compatibility with older equipment that does not support ECN, as they would already set bit 6 to 0

TCP Strategies

Explicit Congestion Notification

- Bit 7, “congestion experienced” (CE) is set by a router when congestion is approaching and the endpoints are ECN-aware
- On receipt of a CE packet, a host should act as if a single packet had been dropped
- So TCP would trigger congestion avoidance and halve the congestion window

TCP Strategies

Explicit Congestion Notification

- TCP also has two bits in the flags field in its header for ECN
- Bits 8 is “ECN echo” (ECE); bit 9 is “congestion window reduced” (CWR)
- Also, TCP options are used in the SYN handshake to negotiate whether the two end are ECN aware

TCP Strategies

Explicit Congestion Notification in TCP

0. a client sends a packet with ECT set in the IP header
1. a router that has congestion imminent sets CE in the IP header
2. the server receives the packet, notes CE is set and sets ECE in the TCP header of the ACK
3. the client reads this and acts as if a single packet has been dropped

TCP Strategies

Explicit Congestion Notification in TCP

1. a router that has congestion imminent sets CE in the IP header
2. the server receives the packet, notes CE is set and sets ECE in the TCP header of the ACK
3. the client reads this and acts as if a single packet has been dropped
4. the clients sets CWR in the TCP header of its next segment to the server to acknowledge that it has received the congestion notification

TCP Strategies

Explicit Congestion Notification

- A technique called *Random Early Detection* (RED) can be used by routers to gauge if congestion is about to happen
- This monitors the lengths of the queues of packets waiting to be relayed forward and notifies congestion when the average queue length exceeds some threshold

TCP Strategies

Explicit Congestion Notification

- Before ECN, the only way to notify congestion was to drop a packet
- ECN has the added benefit of reducing traffic consisting of retransmits of dropped packets
- Theoretically, a fully ECN-aware network would have no loss due to congestion

TCP Strategies

Retransmission Timer

- We now consider the timer that determines when to resend in the absence of an ACK
  - too short a time is poor on slow but otherwise reliable networks
  - too long a time is poor for the data rate
- We want a dynamic behaviour that adapts to changing conditions rather than a simple fixed timeout

TCP Strategies

Retransmission Timer

- If the network slows down (e.g., congestion en route) the timeout should increase
- If the network speeds up (e.g., congestion reduces) the timeout should decrease
- Jacobson gave an easy algorithm: keep a variable, the *round trip time* RTT for each connection

TCP Strategies

Retransmission Timer

- RTT is the best current estimate for the time of a segment going out and the ACK returning
- When the segment is sent, a timer starts
- If the ACK returns before the timeout, TCP looks at the actual round trip time M and updates RTT

$$RTT = \alpha RTT + (1 - \alpha)M$$

TCP Strategies

Retransmission Timer

- α is a smoothing factor, usually 7/8
- Thus RTT increases or decreases smoothly as conditions change and doesn't get too upset by the occasional straggler that is excessively late (or early)
- Next, determine a timeout interval given RTT

TCP Strategies

Retransmission Timer

- This should take the standard deviation of the RTT into account: if the measured RTTs have a large deviation it makes sense to have a larger timeout
- True standard deviations are tricky to compute quickly (square roots), so Jacobson suggested using the *mean deviation*

TCP Strategies

Retransmission Timer

- Mean deviation

$$D = \beta D + (1 - \beta)| RTT - M |$$

- D is close to the standard deviation and is much easier to calculate
- A typical value for β is 3/4

TCP Strategies

Retransmission Timer

- The timeout value is set to

$$T = RTT + 4D$$

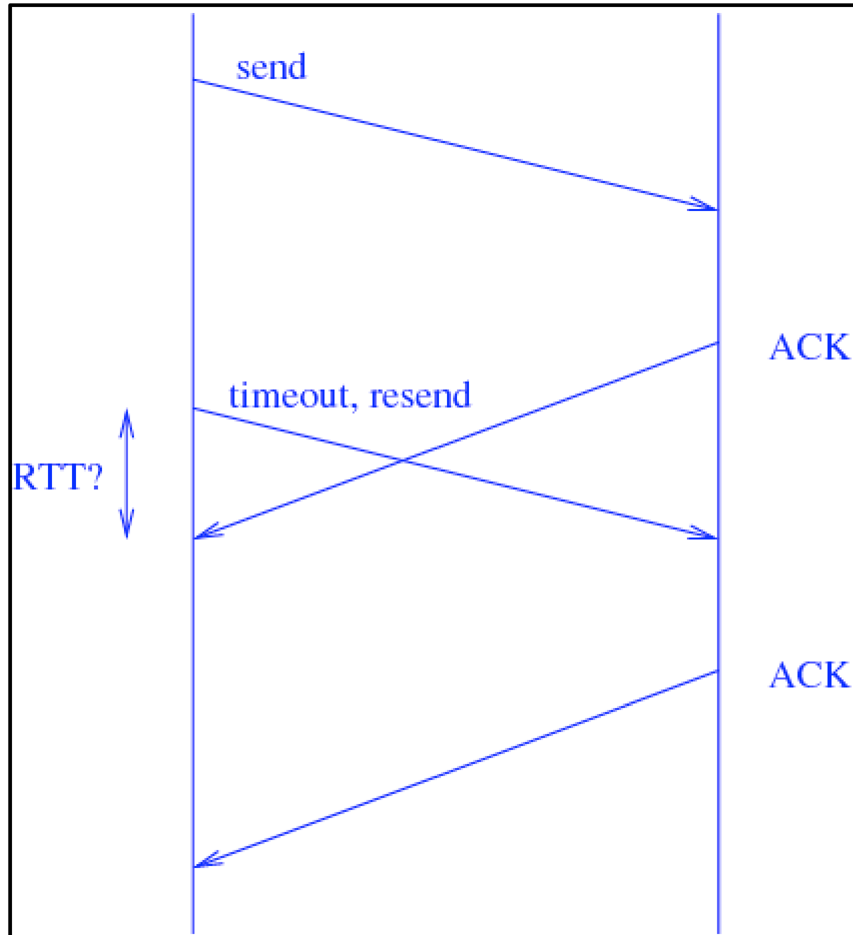
- The 4 was found to be good in practice
- When sending a segment set the timer to expire after time T

TCP Strategies

Retransmission Timer

- What if the timer expires before the ACK?
  - resend the segment
  - update RTT somehow
- Can't use RTT of the resent segment as we might get the ACK of the original segment

TCP Strategies



Retransmission Timer

- This is the *retransmission ambiguity problem*
- The measured RTT would be much too small

TCP Strategies

Retransmission Timer

- Karn's algorithm is to double the timeout T on each failure, but do not adjust RTT
- When segments start getting through normal RTT updates continue and the RTT value quickly reaches the appropriate value
- This doubling is called *exponential backoff*

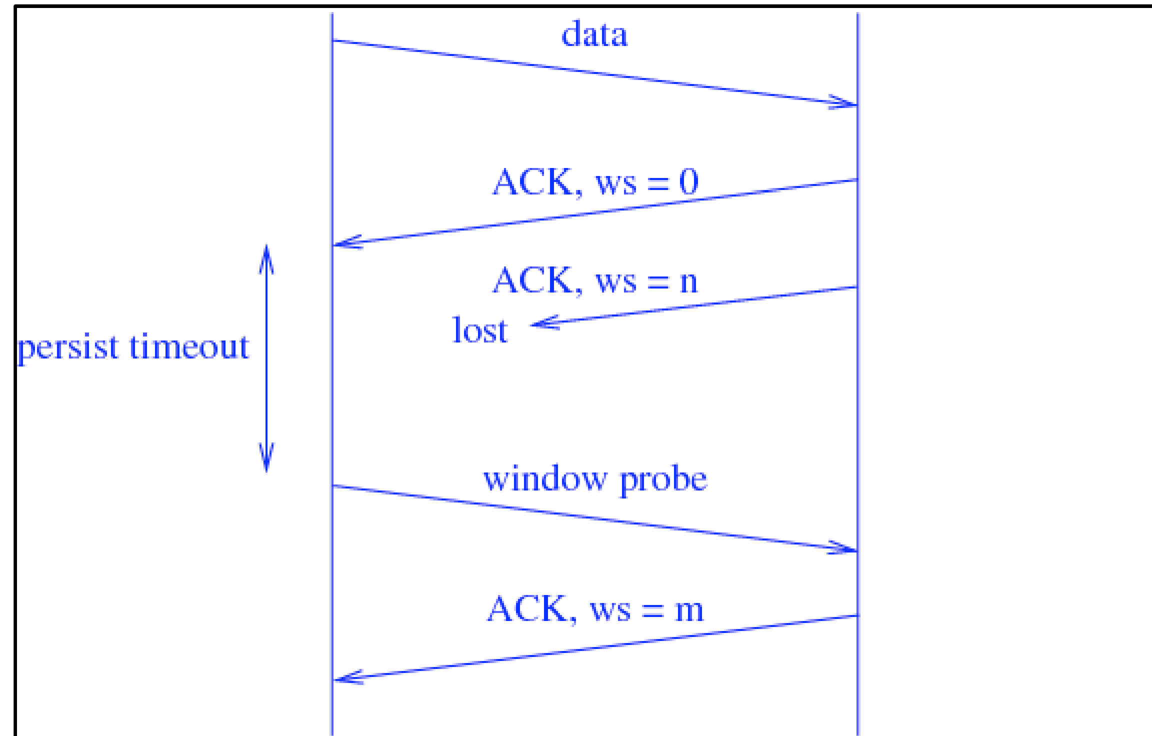
TCP Strategies

Persist Timer

- TCP has several timers. We have seen
  - Retransmission
  - 2MSL
- There is also the *persist timer* (sometimes called the *persistence timer*)
- Its role is to prevent deadlock through the loss of window update segments

TCP Strategies

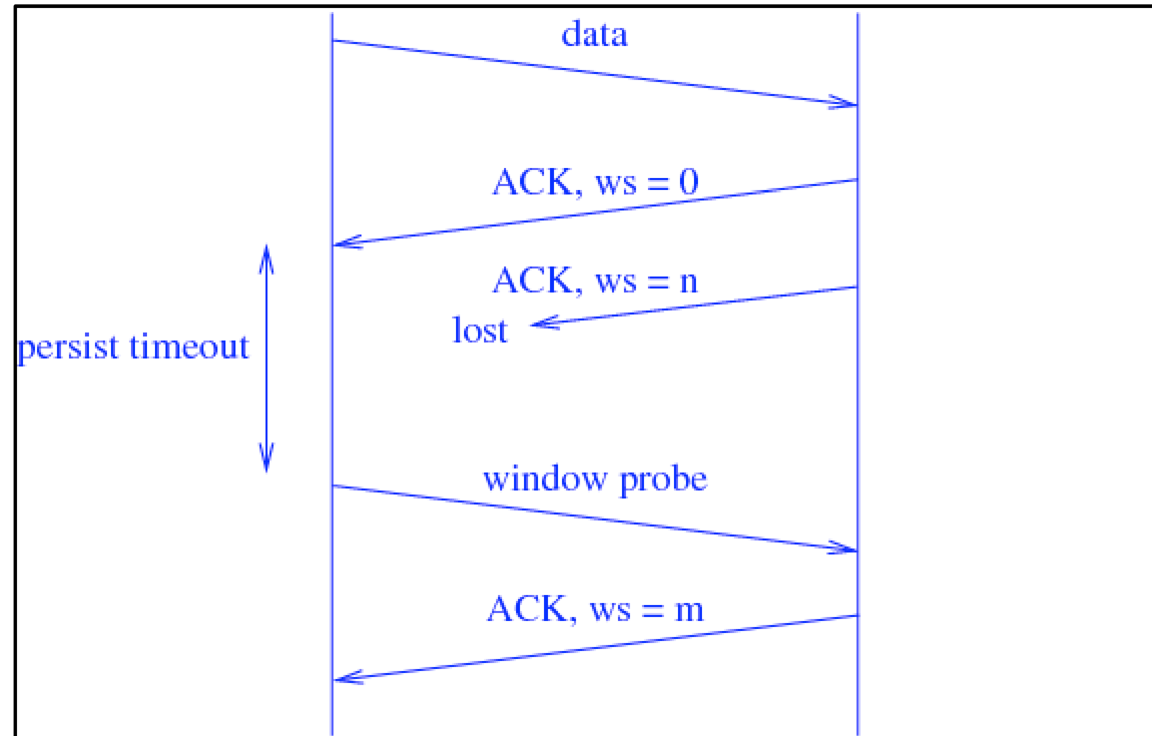
Persist Timer



- A sends to B

TCP Strategies

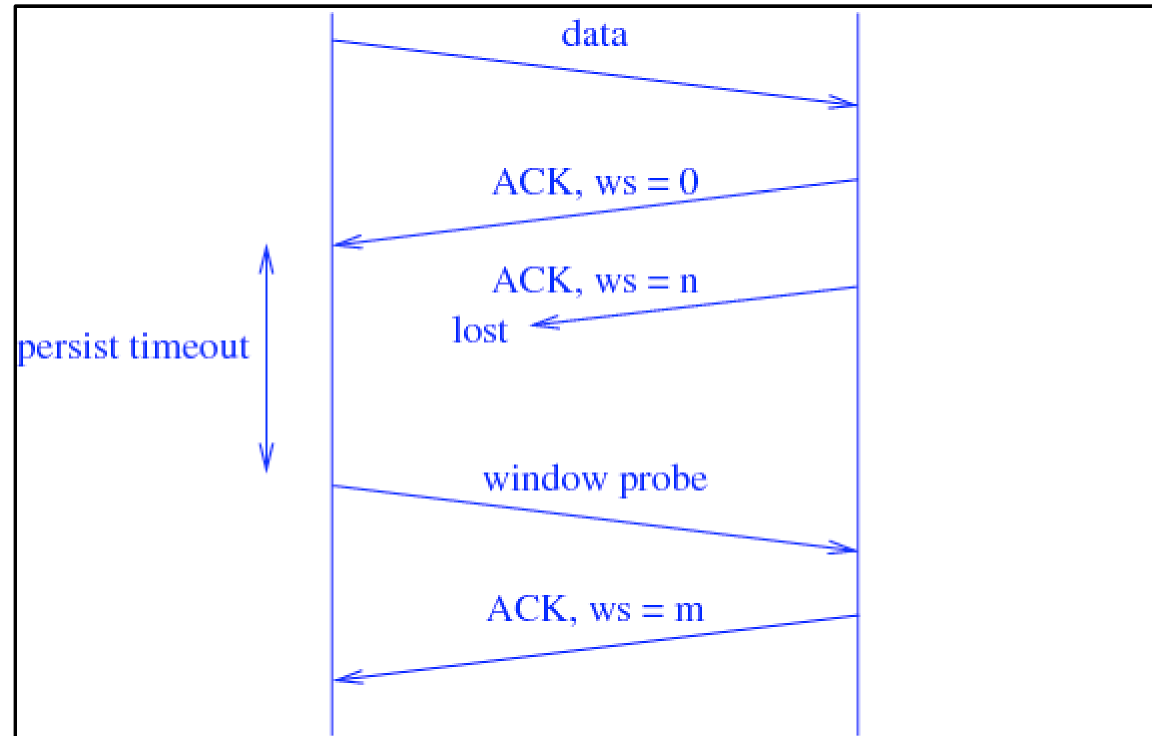
Persist Timer



- B replies with an ACK and a window size of 0

TCP Strategies

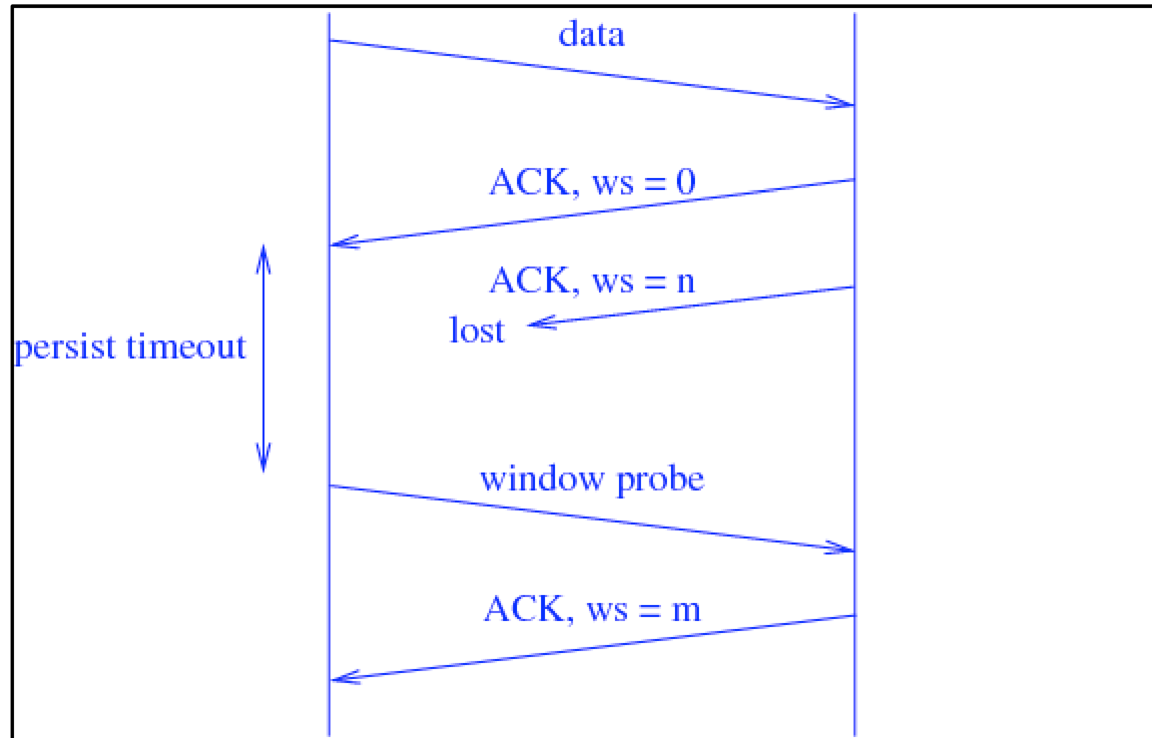
Persist Timer



- A gets the ACK and holds off sending to B

TCP Strategies

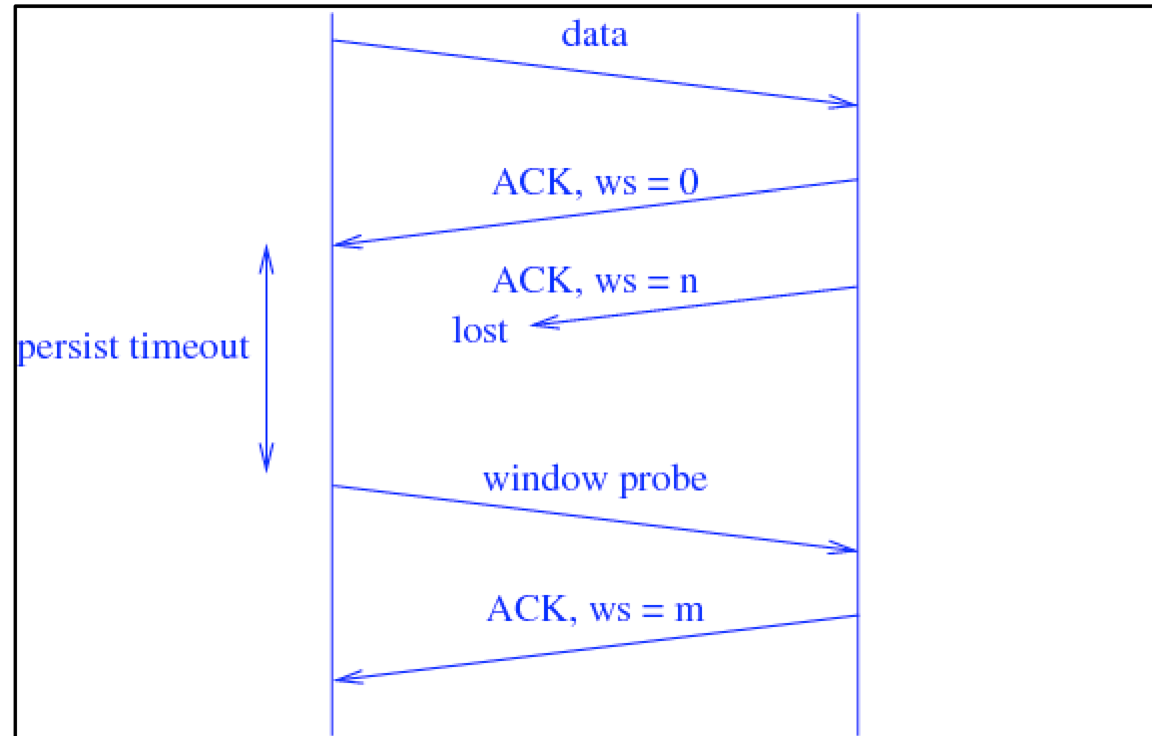
Persist Timer



- B frees up some buffer space and sends a windows update to A

TCP Strategies

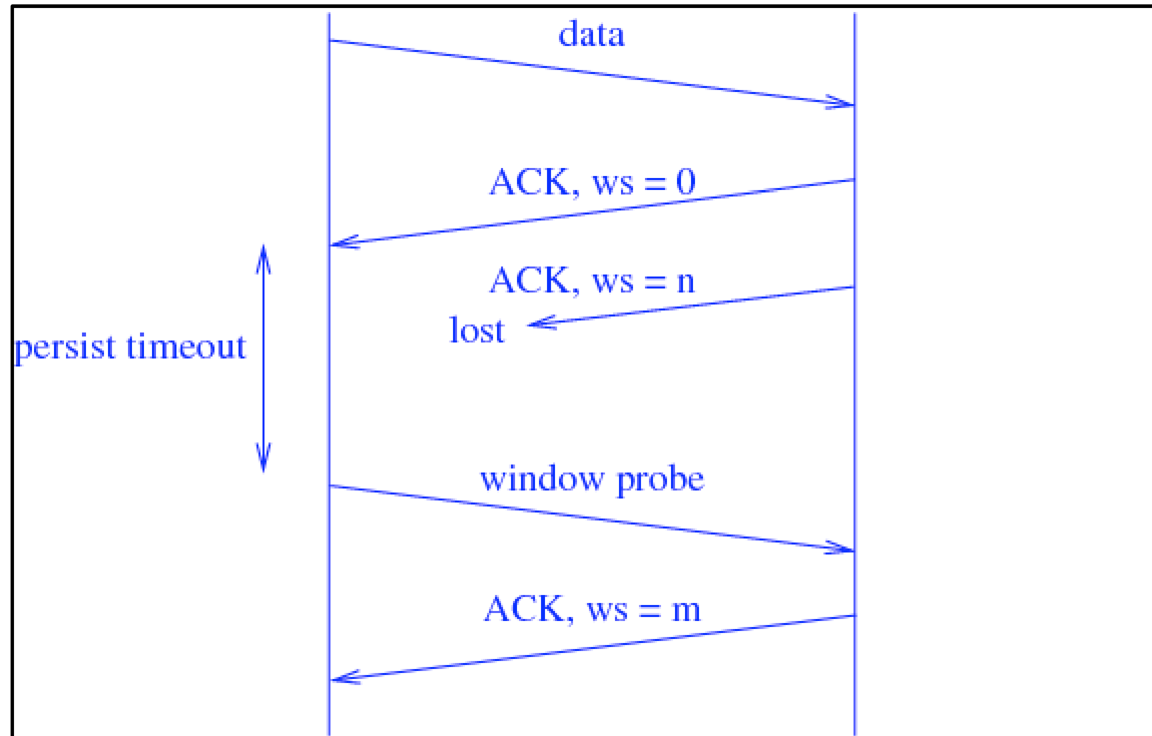
Persist Timer



- This is lost

TCP Strategies

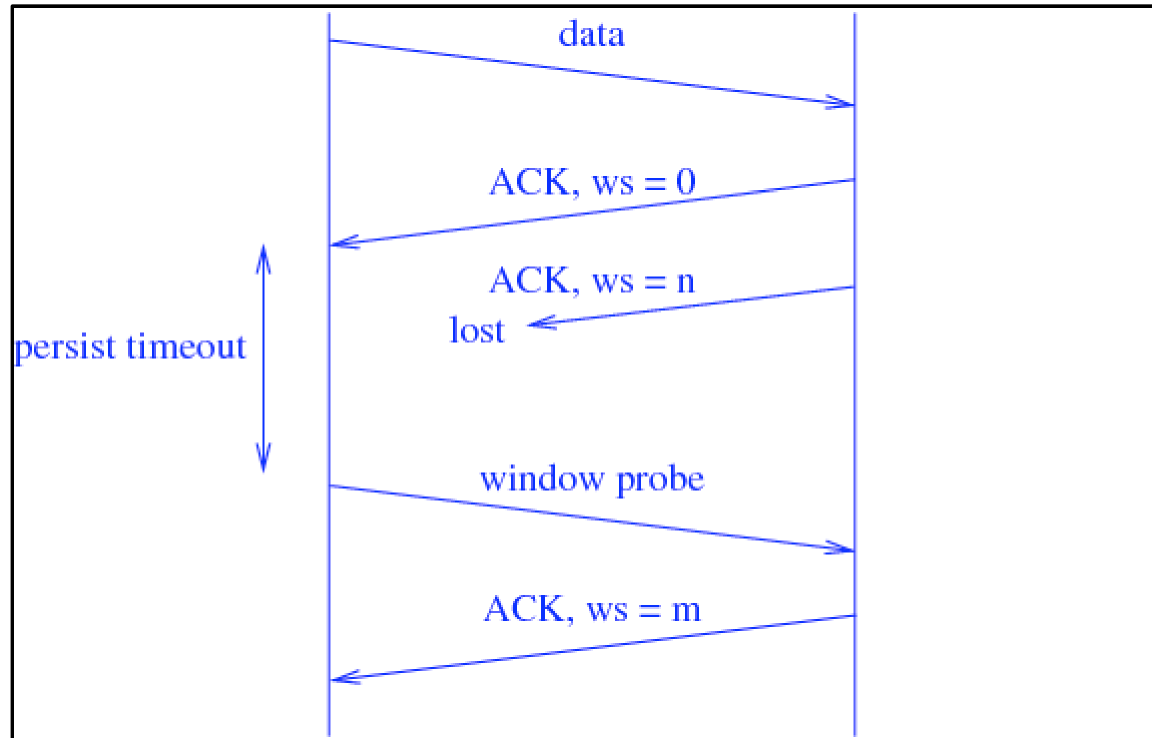
Persist Timer



- Now A is waiting for the window update from B and B is waiting for more data from A: deadlock

TCP Strategies

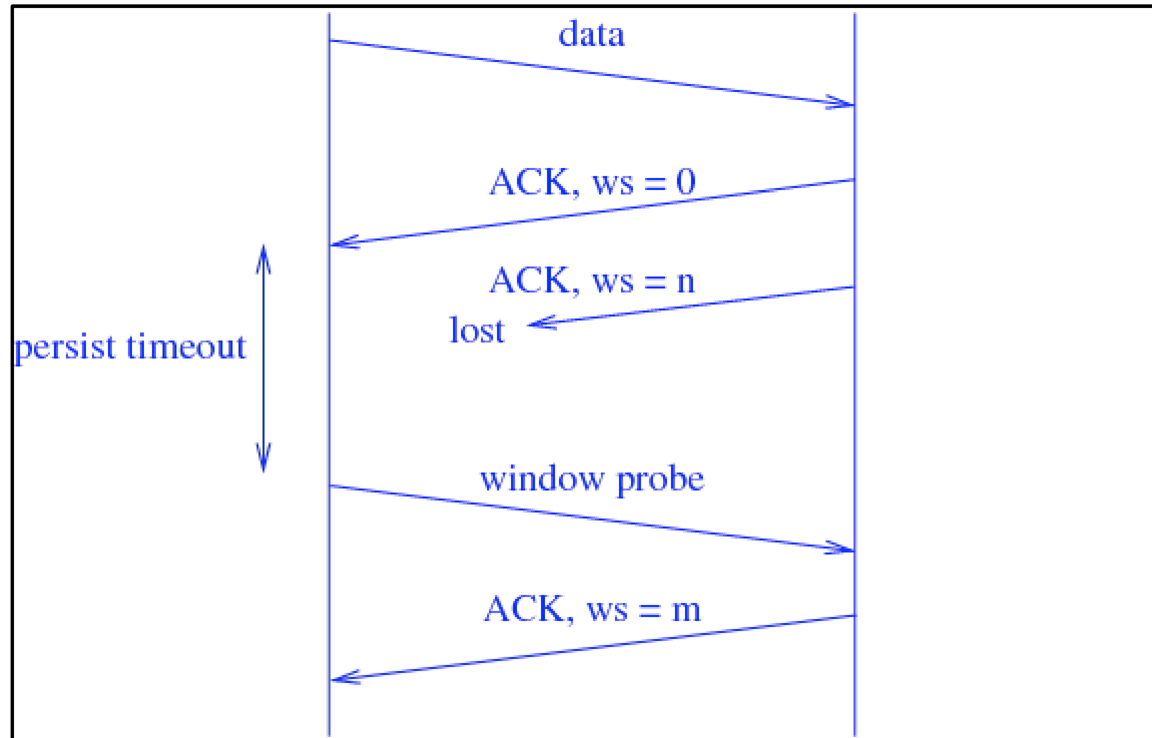
Persist Timer



- To prevent this, A starts the persist timer when it gets the 0 window from B

TCP Strategies

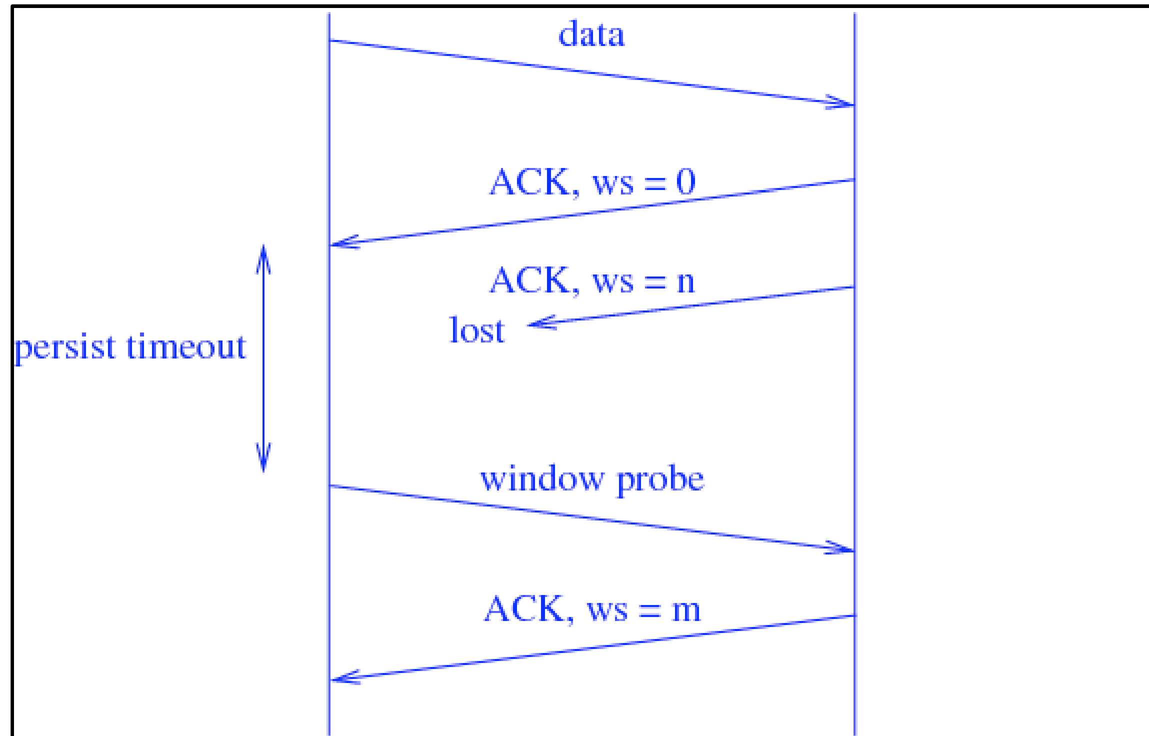
Persist Timer



- If the timer expires, A prods B by sending a 1 byte segment: a *window probe*

TCP Strategies

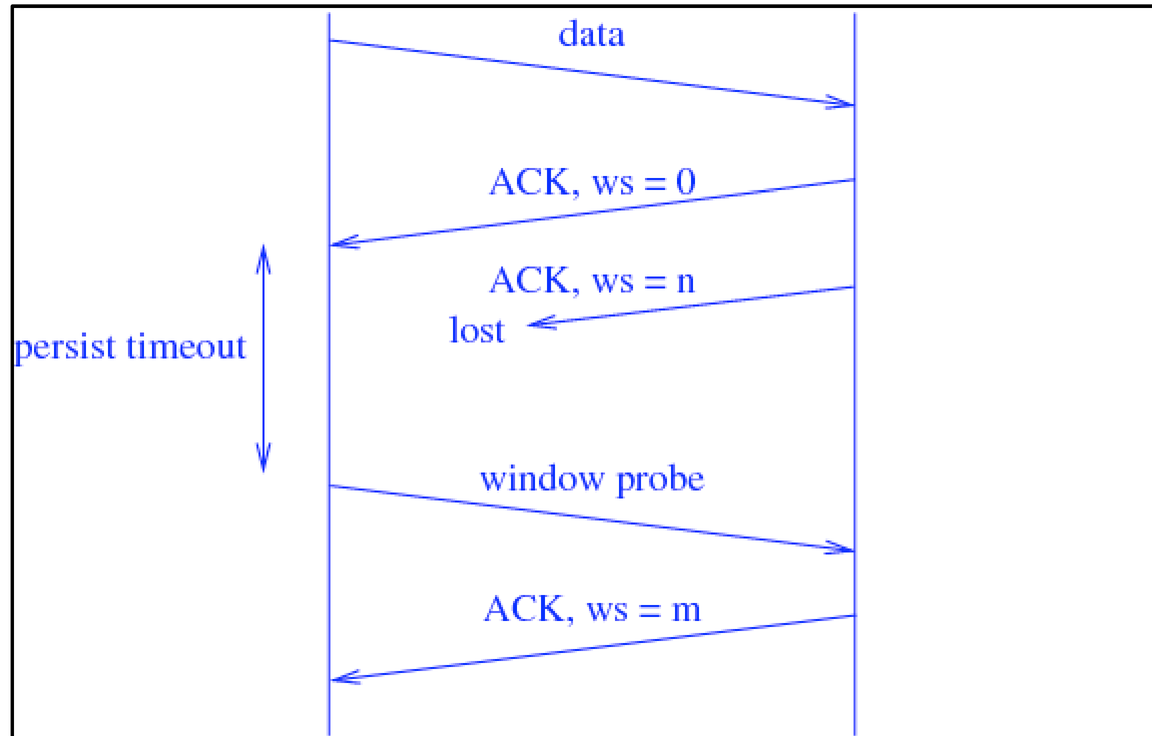
Persist Timer



- If B gets this, the ACK will contain B's current window size

TCP Strategies

Persist Timer



- If the window is still 0, A resets the timer and tries again later

TCP Strategies

Persist Timer

- The persist timer starts at something like 1.5 sec, doubling with each probe and is rounded up or down to lie within 5 to 60 seconds
- So the timeouts are 5, 5, 6, 12, 24, 48, 60, 60, 60, ...
- The persist mechanism never gives up, sending window probes until either the window opens, or the connection closes
- The persist timer is unset when a non-zero window is received