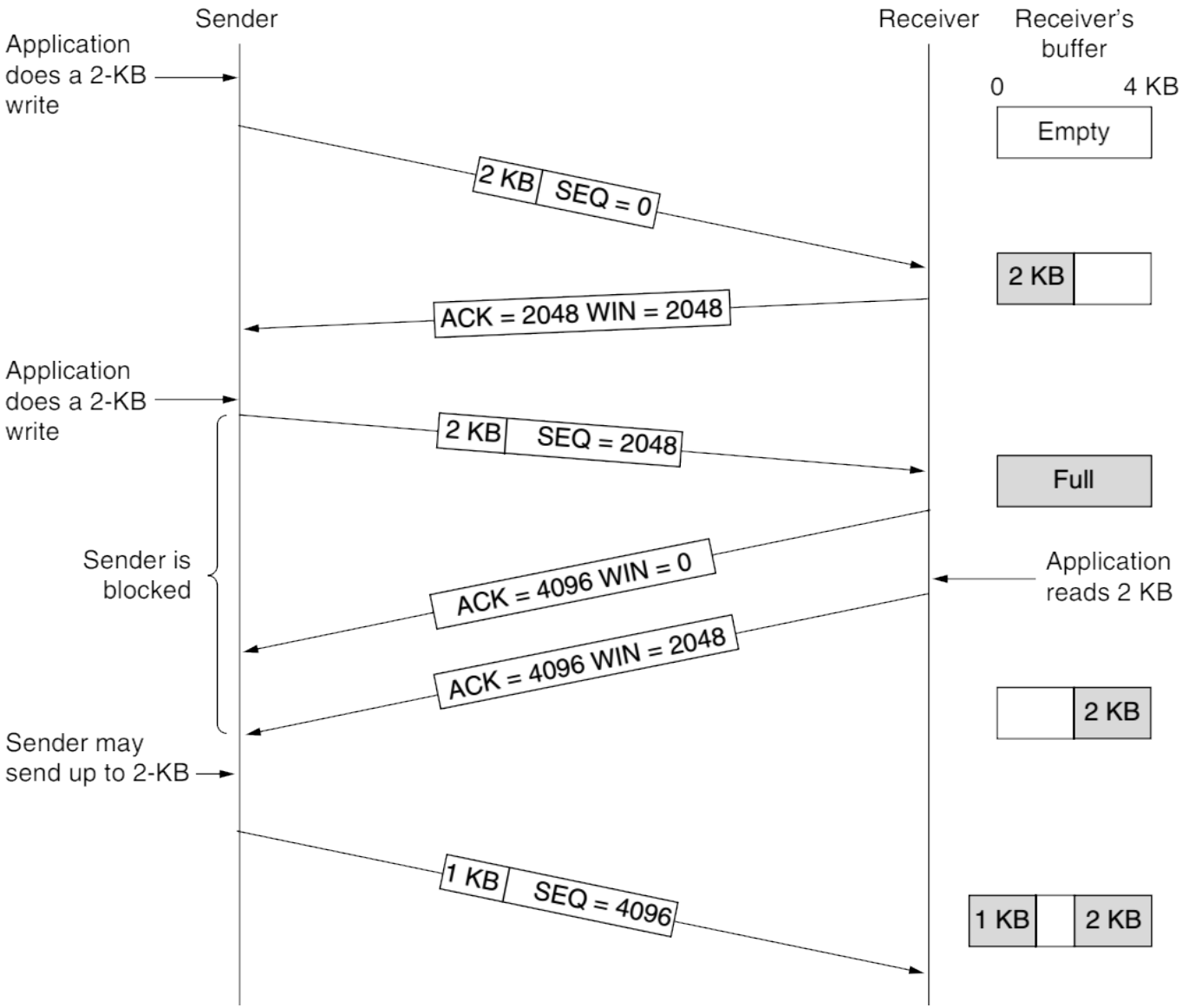
CS3200: Computer Networks

Lecture 28

IIT Palakkad

21 Oct, 2019

TCP Sliding Window



TCP Sliding Window

- When the window is 0, the sender may not normally send segments, with two exceptions.
- First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.
- Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a **window probe**. Why do we need this option?
- Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible.

TCP Sliding Window

Consider a connection to a remote terminal, for example using SSH or telnet, that reacts on every keystroke. Consider the following worst case scenario:

- Whenever a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a **41-byte** IP datagram.
- At the receiving side, TCP immediately sends a **40-byte** acknowledgement (20 bytes of TCP header and 20 bytes of IP header).
- Later, when the remote terminal has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also **40 bytes**.
- When the remote terminal has processed the character, it echoes the character for local display using a **41-byte** packet.

Delayed ACK and Nagle's Algorithm

- One approach that many TCP implementations use to optimize this situation is called **delayed acknowledgements**.
- Delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride.
- What about senders that send multiple short packets? For example, 41-byte packets containing 1 byte of data is still operating inefficiently.
- A way to reduce this usage is known as **Nagle's algorithm** (Nagle, 1984).

Nagle's Algorithms

- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
- In interactive games, players want a rapid stream of short update packets. Gathering the updates to send them in bursts makes the game respond erratically.
- Sometimes interact with delayed acknowledgements to cause a temporary deadlock: the receiver waits for data on which to piggyback an acknowledgement, and the sender waits on the acknowledgement to send more data.
- Nagle's algorithm can be disabled (which is called the TCP NODELAY option).

Silly Window Syndrome

- Occurs when data is passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time.
- Initially, the TCP buffer on the receiving side is full (i.e., it has a window of size 0) and the sender knows this.
- Then the interactive application reads one character from the TCP stream. So, the TCP receiver sends a window update to the sender saying that it is all right to send 1 byte.
- The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment and sets the window to 0.

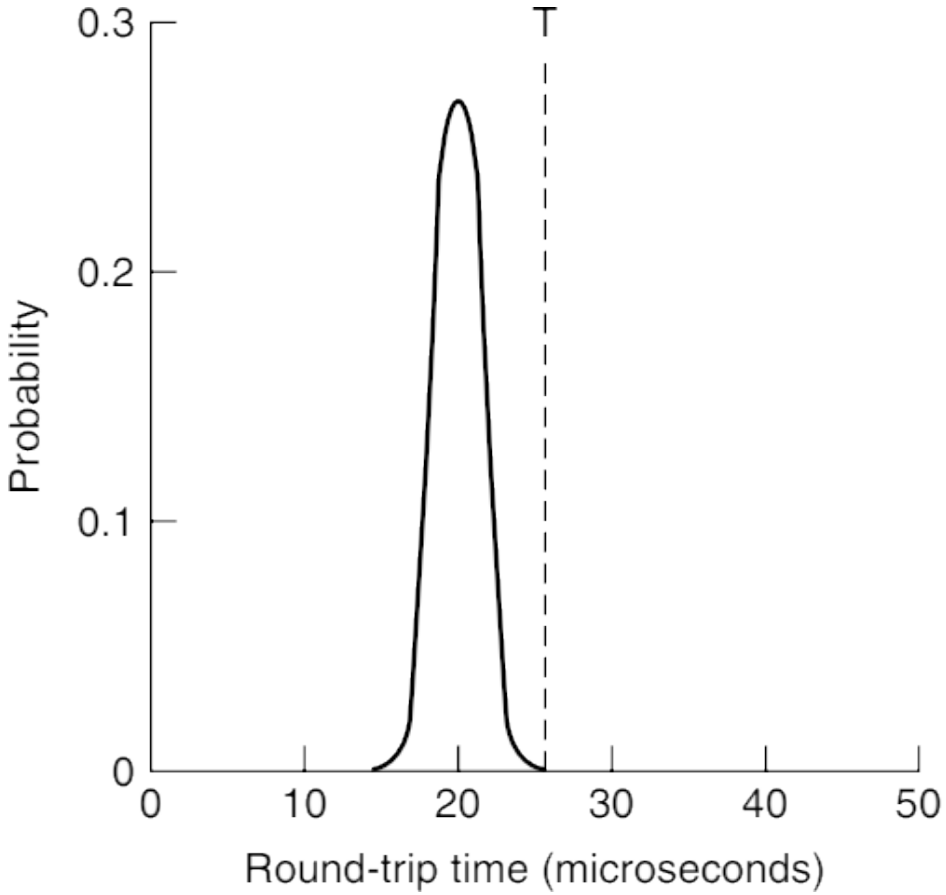
Clark's Algorithm

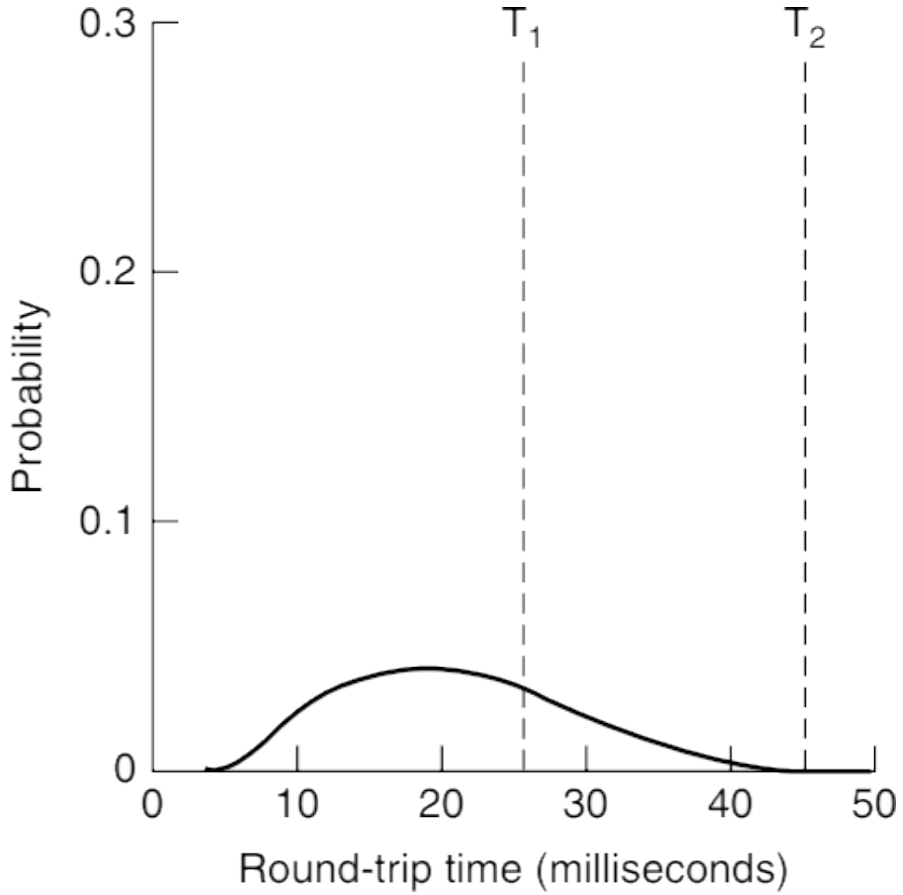
- Clark's solution is to prevent the receiver from sending a window update for 1 byte.
- It is forced to wait until it has a decent amount of space available and advertise that instead.
- Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established or until its buffer is half empty, whichever is smaller.
- Furthermore, the sender can also help by not sending tiny segments. Instead, it should wait until it can send a full segment, or at least one containing half of the receiver's buffer size.

TCP Timer Management

- TCP uses multiple timers (at least conceptually) to do its work.
- The most important of these is the **RTO (Retransmission TimeOut)**.
- When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped.
- If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer os started again). The question that arises is: how long should the timeout be?

TCP Timer Management





Computing RTO

- The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance and is due to Jacobson (1988).
- For each connection, TCP maintains a variable, **SRTT** (Smoothed Round-Trip Time), that is the best current estimate of the round-trip time to the destination in question.
- When a segment is sent, a timer is started, both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, R. It then updates SRTT according to the formula

$$SRTT = \alpha SRTT + (1 - \alpha)R$$

Computing RTO

- Initial implementations of TCP used 2xRTT.
- Queueing models predict that when the load approaches capacity, the delay becomes large and highly variable.
- Jacobson proposed making the timeout value sensitive to the variance in round-trip times as well. This change requires keeping track of another smoothed variable, **RTTVAR** (RoundTrip Time VARiation) that is updated using the formula

$$RTTVAR = \beta RTTVAR + (1 - \beta)|SRTT - R|$$

- The retransmission timeout, RTO, is set to be

$$RTO = SRTT + 4 \times RTTVAR$$

- Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts.

Estimating RTO

- One problem that occurs with gathering the samples, R, of the round-trip time is what to do when a segment times out and is sent again.
- When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one.
- The authors in [Karn and Partridge, 1987] suggested not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time.

Persistence Timer

- A second timer is the **persistence timer**. It is designed to prevent the following deadlock.
- The receiver sends an acknowledgement with a window size of 0, telling the sender to wait.
- Later, the receiver updates the window, but the packet with the update is lost.
- Now the sender and the receiver are each waiting for the other to do something.
- When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still 0, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

Other TCP Timers

- **Keepalive timer:** When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there.

- The last timer used on each TCP connection is the one used in the TIME WAIT state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.