CS3200 Computer Networks **Jul - Nov 2019**

Lecture 28: 21 Oct, 2018

Lecturer: Albert Sunny *Scribe: Sourabh Aggarwal*

This lecture further explains various features, algorithms, mechanisms and timers associated with Transmission Control Protocol (TCP).

28.1 TCP Sliding Window Example

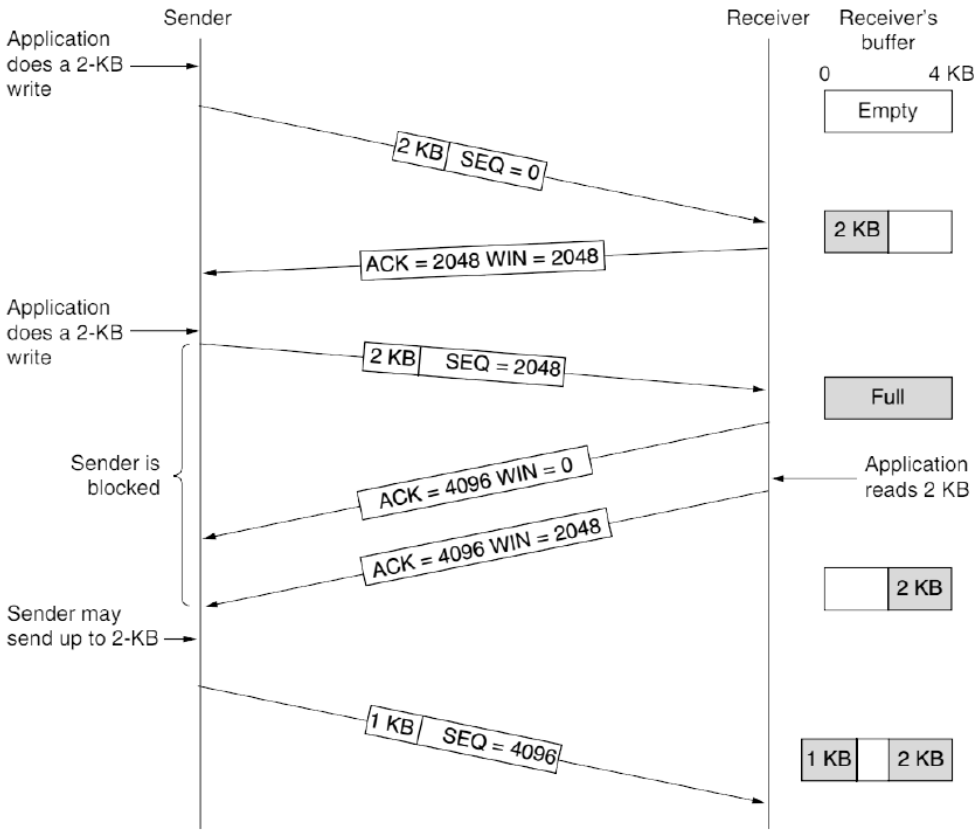

Figure 28.1: TCP Sliding Window Example

Sequence of events:-

1. Application does a 2KB write, sends this data to the receiver, whose buffer is initially empty.

2. Receiver acknowledges this by sending the next expected byte's sequence number (2048) with its current window size (2048).

3. Application again does a 2KB write, sends this data to the receiver making its buffer full.

4. Receiver acknowledges this by sending the next expected byte's sequence number (4096) with its current window size (0).

5. Sender is now stuck until it receives an update from receiver regarding it's window size which it shortly does by again sending acknowledgment and updated window size of 2048.

6. Sending can now send the data, and so on...

When the window is 0, the sender may not normally send segments with two exceptions:-

1. Urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.

2. Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size (Note that in case the receiver's buffer is full, it will reannounce the same thing which it did before). This packet is called a **window probe** and is needed as receiver's window update to the sender might get lost.

Additionally note that 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.

28.2 Delayed ACK

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

1. Whenever a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment (Header is of 20 bytes), which it gives to IP to send as a **41-byte** IP datagram (Again header is of 20 bytes).

2. At the receiving side, TCP immediately sends a **40-byte** acknowledgement (20 bytes of TCP header and 20 bytes of IP header).

3. 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**.

4. When the remote terminal has processed the character, it echoes the character for local display using a **41-byte** packet.

As we can see that we can club steps 2, 3 and 4 together and just send one 41 bytes long packet, reducing total of **162 bytes** to **82 bytes**. This optimisation is called as **delayed acknowledgements** where we delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride.

28.3 Nagle's Algorithm

Previous section's optimisation was on receiver's side but 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). In this algorithm, 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.

Issues with this scheme

- 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.

To remedy these, one can simply disable Nagle's Algorithm via TCP's **NODELAY** option.

28.4 Silly Window Syndrome and Clark's Algorithm

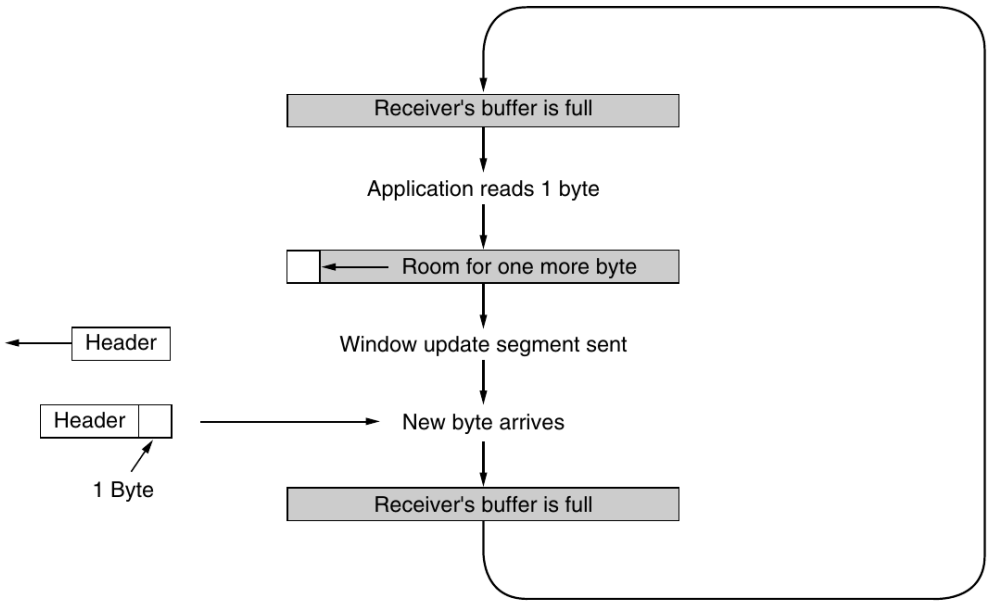

Figure 28.2: Silly Window Syndrome

This problem occurs when data are 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. To see the problem, look at figure above. 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. This action makes the receiving TCP happy, so it 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. This behavior can go on forever.

Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead, 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.

28.5 TCP Timer Management

28.5.1 RTO

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?

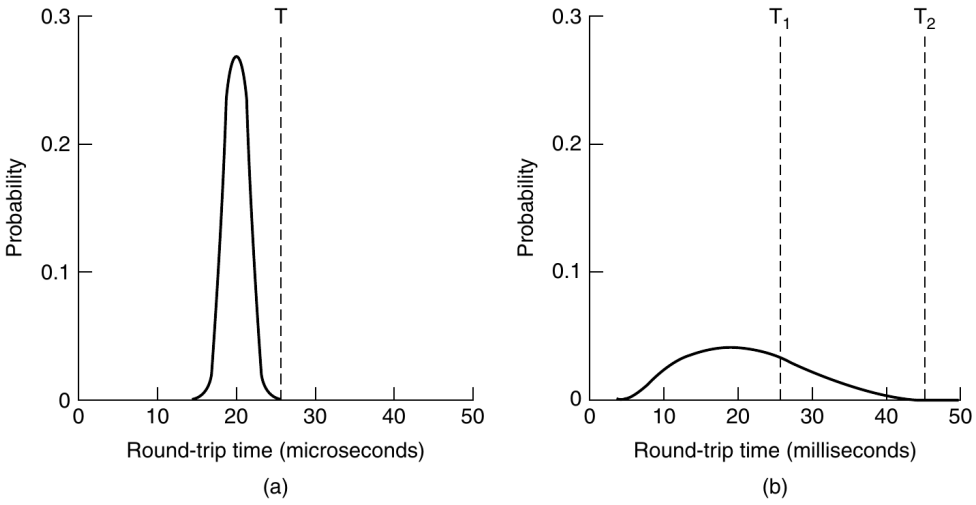

Figure 28.3: (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

This problem is much more difficult in the transport layer than in data link protocols such as 802.11. In the latter case, the expected delay is measured in microseconds and is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 28.3(a). Since acknowledgements are rarely delayed in the data link layer (due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost. TCP is faced with a radically different environment. The probability density function for the time

it takes for a TCP acknowledgement to come back looks more like Fig. 28.3(b) than Fig. 28.3(a). It is larger and more variable. Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say, T_1 in Fig. 28.3(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long (e.g., T_2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

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:-

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

Again updating like this helps in smoothening the change, sort of like a low pass filter that discards noise in the samples. α is called a smoothing factor that determines how quickly the old values are forgotten. Typically, $\alpha = 7/8$. This kind of formula is an **EWMA** (Exponentially Weighted Moving Average)

Initial implementations of TCP used $2 * SRTT$. 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:-

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

This is an EWMA as before, and typically $\beta = 3/4$. The retransmission timeout, RTO, is set to be

$$RTO = SRTT + 4 * RTTVAR$$

The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late. Note that $RTTVAR$ is not exactly the same as the standard deviation (it is really the mean deviation), but it is close enough in practice. Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts.

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 to 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.

28.5.2 Other TCP Timers

- **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.

- **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.