Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

 Lecture 3: TCP Reliability September 15, 1999

Overview. In this lecture, we'll study how transport protocols achieve reliability using TCP as an example. We will look at cumulative acknowledgments (ACKs), the timer algorithms and timeouts, "fast retransmission" and "fast recovery" and selective acknowledgments (SACKs).[1]

Readings. KP91, CJ89[2]

1 TCP reliability

- The Internet is a best-effort network, which implies that
  1. *Losses* occur, typically because of congestion or packet corruption.
  2. Packet *reordering* occurs, typically because of multiple routes between end-points, or bugs in router implementations.
- Many applications want reliable data delivery. Many of these want data *in-order*, i.e., in the order in which the sender sent them.
- On the Internet, such apps use TCP. TCP provides an in-order, reliable, byte-stream (i.e., it doesn't treat datagrams as atomic units, it treats bytes as the fundamental reliability unit) abstraction between two hosts (actually between two interfaces, which isn't exactly the same thing). A TCP connection is *duplex* (bi-directional).
- Reliability achieved through acknowledgments (ACKs). More specifically, the receiver periodically[3] informs the sender about what data it has received via ACKs. TCP ACKs are *cumulative*. For example, the sequence of bytes:

  ```
  1:1000 1001:1700 2501:3000 3001:4000 4001:4576
  ```

  received at the receiver will cause the acknowledgments

  ```
      1001     1701     1701     1701     1701
  ```

  to be sent by the receiver after each datagram arrival. Each ACK acknowledges all bytes received thus far and tells the sender what the next expected in-sequence byte is.

---

[1]The original intention was also to study the principles behind Internet congestion management and increase/decrease mechanisms. We ran out of time partly because of the digression we took into MPLS, IPv6 flow labels, IPv6 deployment issues, and so on.

[2]Because of footnote 1, this is reading for next class.

[3]Every time it receives a datagram; modern TCP receivers implement a *delayed ACK* policy where they ACK only on every other received datagram as long as data arrives in sequence, or every 500ms, whichever is smaller. BSD-derived implementations use a 200ms "heartbeat" timer for this.

© 1999 Hari Balakrishnan DO NOT REDISTRIBUTE

- Each ACK includes in it a receiver-advertised window that tells the sender how much space is available in its socket buffer at any point in time. This is used for end-to-end *flow control.* This is not to be confused with *congestion control,* which is how resource contention for "inside-the-network" router resources (bandwidth, buffer space) is dealt with. Flow control only deals with making sure that the sender doesn't overrun the receiver at any stage (it's a lot easier that congestion control, as you can see).
- With cumulative ACKs, TCP has two loss recovery modes—*timeouts* and *fast retransmissions.*

1.1 TCP timers

To perform retransmissions, the sender needs to know when packets are lost. If it doesn't receive an ACK within a certain amount of time, it assumes that the packet was lost and retransmits it. This is called a *timeout* or a *timeout-triggered* retransmission. The problem is to determine how long the timeout period should be.

What factors should the timeout depend on? Clearly, it should depend on the connection's round-trip time (RTT). To do this, the sender needs to estimate the RTT. It obtains samples by monitoring the time difference between sending a segment and receiving a positive ACK for it. It needs to do some averaging across all these samples to maintain a (running) average. There are many ways of doing this; TCP picks a simple approach called the Exponential Weighted Moving Average (EWMA), where $srtt = \alpha \times r + (1 - \alpha) srtt$. Here, r is the current sample and $srtt$ the running estimate of the *smoothed* RTT. $\alpha = 1/8$ in TCP for efficient computability (can be implemented in fixed point using bit-shifts).

We now know how to get a running average of the RTT. How do we use this to set the retransmission timeout, or RTO? One option, an old one, is to pick a multiple of the smoothed RTT and use it. For example, we might pick $RTO = \beta * srtt$, with β set to a constant like 2. In fact, the original TCP specification (RFC 793) used precisely this method. Unfortunately, this simple choice doesn't work too well in preventing *spurious* retransmissions. Spurious retransmissions are those done for segments still in transit but presumed lost by the sender. (This could lead to bad congestion effects, as we'll see next lecture.)

A nice and simple fix for this is to make the RTO a function of both the average and the standard deviation. Under nice Gaussian assumptions of delay samples (not true in practice, but it still works), the tail probability of a spurious retransmission when the RTO is a few standard deviations away from the mean is rather small. So, TCP uses an RTO set according to: $RTO = srtt + 4 \times rttvar$, where $rttvar$ is the (inappropriately named) mean linear deviation of the RTT from its mean. I.e., $rttvar$ is calculated as $rttvar = \gamma \times dev + (1 - \gamma) \times rttvar$, where $dev = |r - srtt|$ and $\gamma = 1/4$.

This isn't the end of the story. As Karn's paper describes, TCP suffers from a significant *retransmission ambiguity* problem. When an ACK arrives for a segment the sender has retransmitted, how does it know whether the RTT to use is for the original transmission or for the retransmission? This might seem like a trivial detail, but in fact is rather vexing because the RTT estimate can easily become meaningless and throughput can suffer. To solve this problem, Karn came up with a surprisingly simple fix—simply ignore samples that arrive when a retransmission is pending or in progress. I.e., don't consider samples for any segments that have been retransmitted.

The modern way of avoiding the ambiguity problem is to use the TCP timestamp option. Most good TCP implementations follow RFC 1323's recommendation of using the timestamp option. Here, the sender uses 8 bytes (4 for seconds, 4 for microseconds) and stamps its current time on the segment. Teh receiver, in the cumulative ACK acknowledging the receipt of a segment simply *echoes* the sender's stamped value, and now the sender can do the calculation trivially by subtracting teh schoed time in the ACK from the current time. Note that it now doesn't matter if this was a retransmission or an original transmission.

The other important issue is deciding what happens when a retransmission times out. Obviously, because TCP is a "fully reliable" end-host protocol, the sender must try again. But rather than try at the same frequency, it takes a leaf out of the contention protocol literature and performs exponential backoffs of the retransmission timer.

A final point to note about timeouts is that they are extremely *conservative* in practice. TCP retransmission timers are usually (but not always) coarse, with a granularity of 500 or 200 ms. This is a big reason why spurious retransmissions are rare in modern TCPs, but also why timeouts during downloads are highly perceptible by humans.

1.2 Fast retransmissions

Because timeouts are expensive (in terms of killing throughput at the expense of network stability), it makes sense to explore other retransmission strategies that are more responsive. Such strategies are also called *data-driven* (as opposed to *timer-driven*) retransmissions. Going back to the earlier example:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

with ACKs

```
      1001      1701      1701      1701      1701
```

It's clear that a sequence of repeated ACKs are a sign that something strange is happening, because in normal operation cumulative ACKs should monotonically increase. Repeated ACKs in the middle of a TCP transfer can occur for three reasons:

1. Window updates. When the receiver finds more space in its socket buffer, because the application has read some more bytes, it can send a window update even if no new data has arrived.
2. Segment loss.
3. Segment reordering. This could have happened, for example, if datagram 1701-2500 had been reordered because of different routes for different segments.

Repeated ACKs that aren't window updates are called *duplicate* ACKs or *dupacks*. TCP uses a simple heuristic to distinguish losses from reordering: if the sender sees an ACK for a segment more than three segments larger than a missing one, it assumes that it's been lost. Unfortunately, cumulative ACKs don't tell us which segments have reached; they only tell the sender what the last in-sequence byte was. So, the sender simply *counts* the number of dupacks and infers that if it see three or more dupacks, that the corresponding segment was lost. Various empirical studies have shown that this heuristic works pretty well, at least on today's Internet, where reordering TCP segments is strongly discouraged and there isn't much "parallel" ("dispersity") routing.

1.3 Selective acknowledgment (SACK)

When the bandwidth-delay product of a connection is large, e.g., on a high-bandwidth, high-delay link like a satellite link, windows can get pretty large. When multiple segments are lost in a single window, TCP usually times out. This causes abysmal performance for such connections, which are colloquially called “LFNs” (for “Long Fat Networks” and pronounced “elephants,” of course). Motivated in part by LFNs, selective ACKs (SACKs) were proposed as an embellishment to standard cumulative ACKs. SACKs were standardized a few years ago by RFC 2018, after years of debate.

Using the SACK option, a receiver can inform the sender of up to three maximally contiguous blocks of data it has received. For example, for the data sequence:

```
1:1000 1001:1700  2501:3000  3001:4000 4577:5062 6000:7019
```

received at the receiver will cause the ACKs and SACKs (in brackets)

```
      1001      1701        1701         1701       1701         1701
                        [2501-3000]  [2501-4000] [4577-5062; [6000-7019;
                                                  2501-4000]  4577-5062;
                                                              2501-4000]
```

SACKs allow LFN connections to recover many losses in the same window with a much smaller number of timeouts. While SACKs are in general a Good Thing, they don’t always prevent timeouts, including in some situations that are common in the Internet today. [Why not?]

1.4 Some other issues

- **Segment size.** How should TCP pick its segment size for datagrams? Each side picks a “default” MSS (maximum segment size) and exchanges the the SYN exchange at connection startup; the smaller of the two is picked.

  The recommended way of doing this, to avoid potential network fragmentation, is via *path MTU discovery.* Here, the sender sends a segment of some (large) size that’s smaller than or equal to its interface MTU (when the IP and link-layer headers are added) with the “DON’T FRAGMENT (DF)” flag in the IP header. If the receiver gets this segment, then clearly every link en route could support this datagram size because there was no fragmentation. If not, and an ICMP error message was received by the sender from a router saying that the packet was too big and would have been fragmented, the sender tries a smaller segment size until one works.

- **Low-bandwidth links.** Several TCP segments are small in size and are mostly comprised of headers; examples include telnet packets and TCP ACKs[4]. To work well over low-bandwidth links, TCP header compression can be done (RFC 1144). This takes advantage of the fact that most of the fields in a TCP header are either the same or are predictably different from the previous one. This allows a 40-byte TCP+IP header to be reduced to as little as 3-6 bytes.

[4] TCP does allow data to be piggybacked with ACKs and most data segments have valid ACK fields acknowledging data in the opposite direction of the duplex connection.