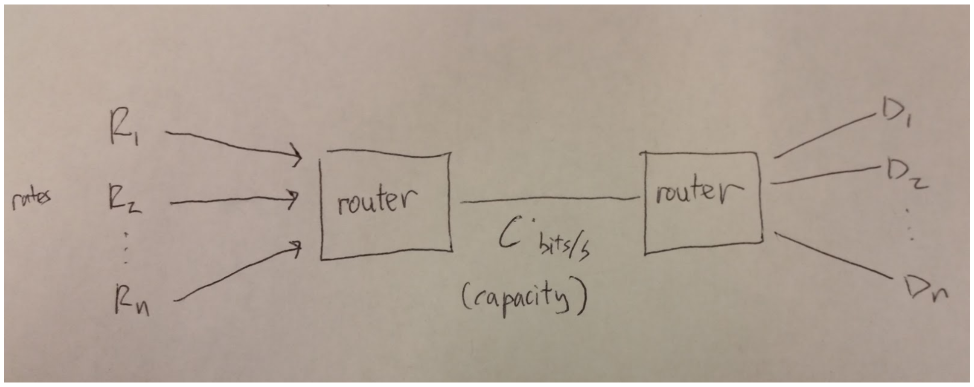
6.033 Spring 2015
Lecture #11: Transport-Layer Congestion Control
Hari Balakrishnan
Scribed by Qian Long

Please read Chapter 19 of the 6.02 book for background, especially on acknowledgments (ACKs), timers, retransmissions, and the sliding window protocol. This lecture will assume you know all that. We will also assume that you know the causes for delays in packet-switched networks: propagation, processing, transmission, and queueing. These are in Chapter 16. One of the key ideas in congestion control is intellectually similar to backoffs in media-access (MAC) protocols, which are described in Chapter 15. Chapters 15 and 16 also describes key notions like network utilization and fairness (equitable resource allocation), which we will use here without defining them formally. A good understanding of all these ideas will make it a lot easier to appreciate what we're learning in 6.033.

The Internet is a best effort network: packets experience losses, variable delays, reordering, duplication, etc. We want to abstract all these things away from the application.That's what TCP does. There are many possible transport protocols on the Internet, but TCP is the most common. Viewed as a software program, it is probably the most widely deployed program in the world today!

What is TCP?
It stands for the Transmission Control Protocol. It provides:
- an in order, reliable byte stream
- congestion control (main focus today)

What is congestion?
- informal definition: rate of jobs/packets/"stuff" arriving > rate of service



Suppose we have *N* sources, with source *i* transmitting packets at rates R_i bits/s via a router with a C bits/s link, as shown in the picture above. Then, congestion is said to occur when

$$\sum_{1}^{N} R_i > C \qquad (1)$$

This definition is missing a time-scale over which we measure the rates. because this doesn't capture the duration of the congestion (i.e. congestion happening in bursts when there are bursts of traffic)

Our goal is to develop a *distributed* protocol (without any centralized controller) running at the sources, with cooperation from the destinations and some information gleaned from the routers, to select the rates R_i . As network conditions change, we expect the source rates R_i to also change and *adapt*.

There is one natural time-scale in this setting: the round-trip time, or RTT, of the connection. Recall the definition of the RTT: it is the time between when the sender transmits a packet and when it receives an ACK for it from the receiver. The "typical" RTT on a wide-area Internet path is 100 milliseconds, although the number can vary these days from a few microseconds (in datacenter networks) to hundreds of milliseconds (satellite links).

- time from when sender sends a packets to when they receive an ACK

Let's multiply both sides of the congestion condition (1) by a time-scale of measurement, *T*:

$$\sum_{1}^{N} R_i * \ T > C * T \qquad (2)$$

There are 3 regimes of *T* that are interesting to us:

1. T < 1 RTT (< ~100 ms)
2. T between 1 and 10000 RTT (approx 100 ms to 1000 seconds)
3. T > 10000 RTT (> ~1000 seconds)

Case (3) is a longer-term *provisioning* issue, where congestion persisting over such long time-scales must be handled by adding additional resources (bandwidth). We will tackle cases (1) and (2) in this lecture (and also the next).

Case (1) requires some support from routers. At the very least, the router should provide some queuing to absorb bursts of traffic. A useful rule of thumb is that a router's queue length should be on the order of 1 RTT or less to achieve high utilization; i.e., its size should be equal to the product of the RTT expected for connections (excluding queueing delays) and the link's data rate. This quantity is . Anything more is unnecessary because the endpoints can hear from the receivers in about 1 RTT, and can adapt. It's really only for congestion over shorter time scales that we need buffering in the router. A situation with buffering far exceeding this quantity has long been known to be a problem; in recent years, the colorful term "bufferbloat" has been used to describe the phenomenon, observed on certain network devices that have an excess of buffering in them (perhaps because RAM is inexpensive and some network engineers *wrongly* view packet drops as an evil to be avoided).

In fact, if there are multiple concurrent flows sharing the network link and they are not all synchronized with each other, it is unlikely that they will all burst traffic at the same time. In that case, it turns out that an amount of buffering equal to the bandwidth-delay product divided by the square-root of the number of flows suffices to achieve high network utilization.

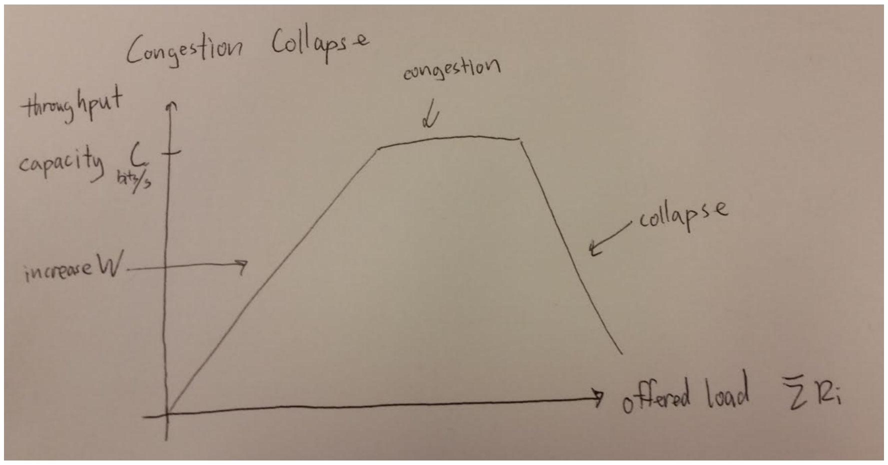
Our approach to congestion control builds on the sliding window protocol.

Sliding Window

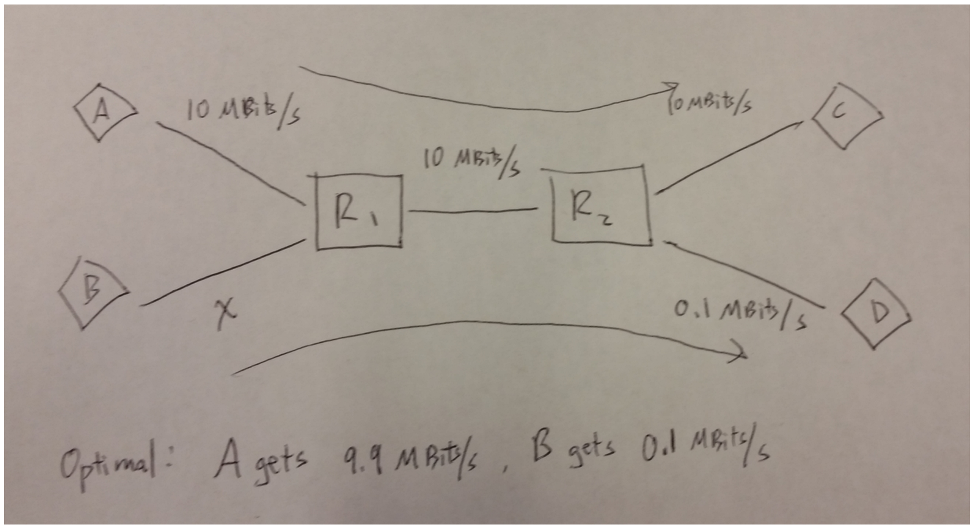
- Invariant: there are W unacknowledged packets in flight
- If the round trip time is RTT, then the rate, or throughput is W / RTT
- W affects the transmission rate, we want to know how to set W. Adjusting W dynamically has the effect of adjusting the rate dynamically
- We could also directly *pace* packets out using a timer. But using a sliding window with a variable size has the beautiful property that ACKs "pace" packets out and act as a natural "clock" for sending packets. This concept called "ACK clocking" or "self-clocking". Although a timer-based approach has certain benefits (which we won't delve into here), ACK clocking has the property that if the network suddenly got congested, the sender would react immediately, because the rate at which ACKs arrive would immediately slow down, tracking the sudden onset of congestion. The ACK clock, which is a benefit of the sliding window method, provides a natural way to handle sudden congestion events in a safe way.

Why do we need congestion control?
In the extreme case, without it, we run the danger of **congestion collapse**.

- If an increase in offered load causes the throughput to *decrease*, the network is said to suffer congestion collapse. It is illustrated below.



- A simple example is shown in the following diagram (B sends at *X* Megabits/s, and the link between B and R1 has a capacity of 10 Megabits/s).



- If we set *X* to 0, then A gets 10 and B gets 0. If we set *X* to 10, then A gets 5 Megabits/s and B gets 5 Megabits/s (due to the bottleneck to D, assuming R1 gives A and B each half the bandwidth).
- This is an example of a congestion collapse: as the offered load, in this case *X*, increases, the total throughput, in this case the sum of the receiver throughputs at C and D, *decreases*.

We need a way for the senders to pick a good window size.

Basic Congestion Control Algorithm: Strategy

On congestion (dropped packets, etc): reduce window
On no congestion: increase window

Additive Increase Multiplicative Decrease (AIMD)
- This is an example of a *linear* algorithm
- Note: *cwnd* is the window size (denoted as *W* in the above section)
- General window size equation: $cwnd \leftarrow \alpha * cwnd + \beta$
- Algorithm (example):
  - on congestion: set $\alpha = \frac{1}{2}$ and $\beta = 0$, so

    $$cwnd \leftarrow cwnd / 2$$

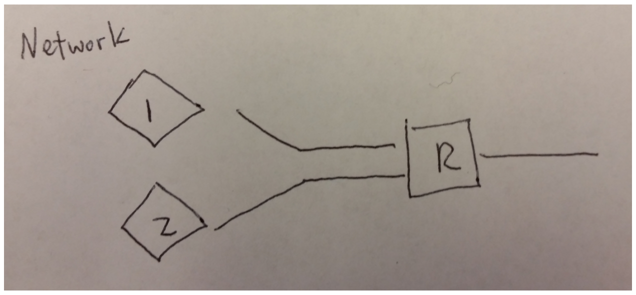
    This step is multiplicative decrease (MD)
  - on no congestion: set $\alpha = 1$ and $\beta = 1$, so, on each RTT,

    $$cwnd \leftarrow cwnd + 1$$

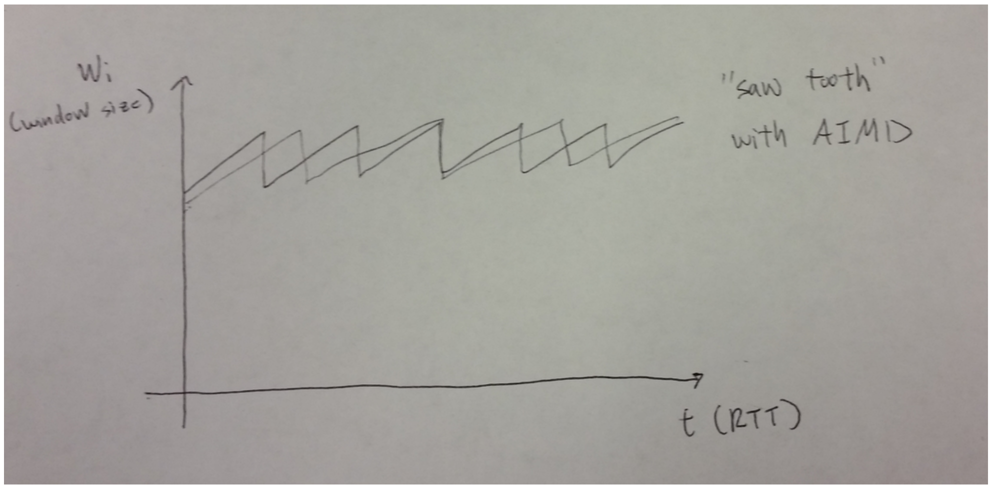
This step is additive increase (AI)

- Why AIMD? It's a "safe algorithm". It is able to achieve two properties:
  - efficiency (utilization)
  - equitability (fairness)

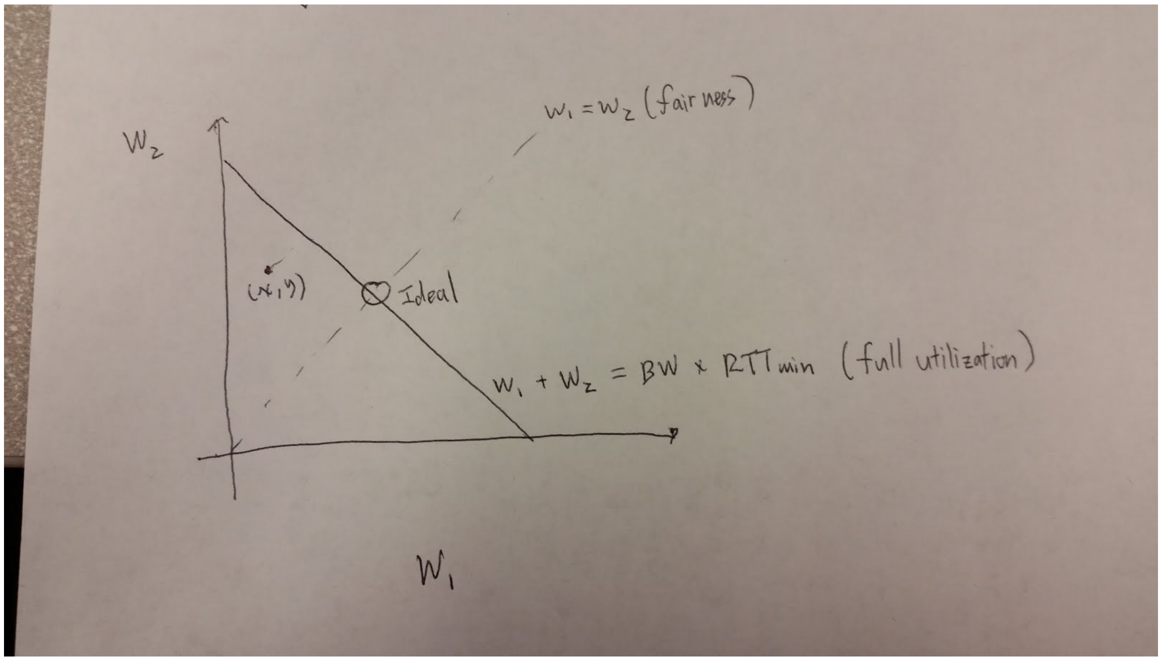
- Intuition for why it works
  - consider a simple 2 sender network where the two senders receive *synchronized feedback* about congestion; i.e., when the network is congested,both senders hear about it at the same time, and likewise when the network is uncongested.



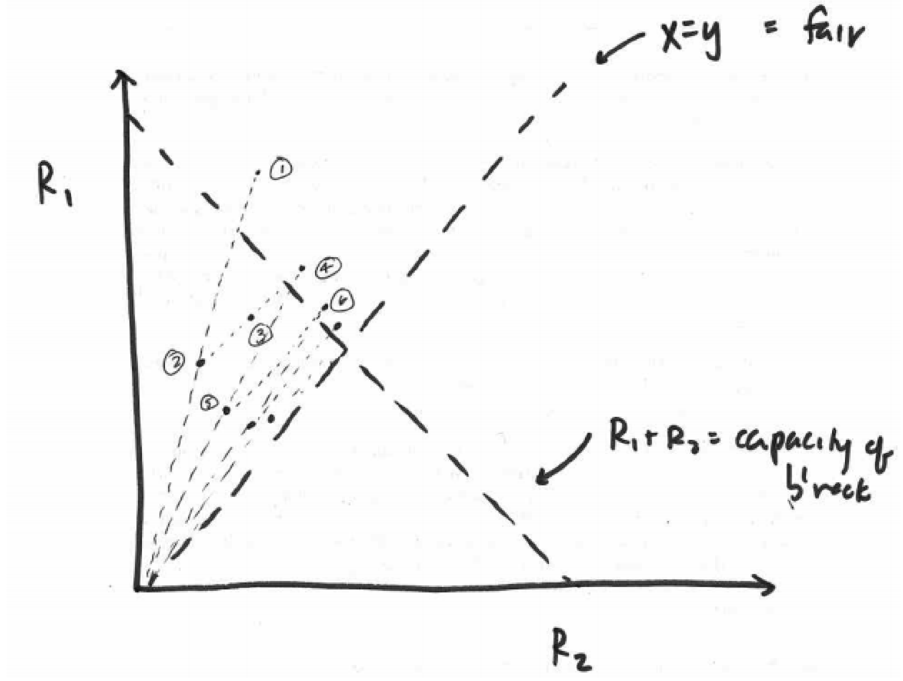
- Plotting both window sizes vs time, you get a "saw tooth" graph when using AIMD



- If we were to eliminate time and plot W2 against W1, we get the following "phase space" diagram:

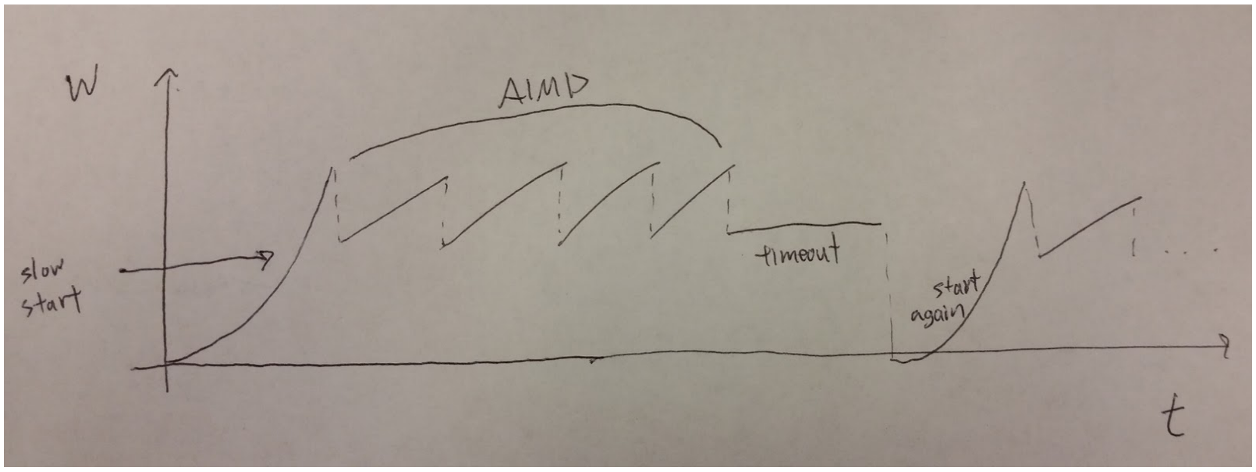


- The point at the intersection of the two lines is the ideal, achieving both fairness and utilization.
- Consider the point (x, y) shown above. Since that point is under utilizing bandwidth, both senders will additively increase their window sizes, leading to (x + 1, y+1), which is a 45 degree increment towards the utilization line. That keeps occurring until we get to a point (x',y') above the "full utilization" line. Here, there is congestion, so both senders multiplicatively decrease, and the point would becomes (x'/2, y'/2), on the line toward the origin. If one continues this procedure, the scheme converges to the "ideal" point shown above.



Problems with AIMD

- increases very slowly at the beginning
- initial window size is 1 (probably too small in practice, companies have different opinions on good window size)
- solution is to do **multiplicative increase** at the beginning to ramp up
    - $cwnd_{init} = 1$
    - initially, do $cwnd \leftarrow 2 * cwnd$ each RTT until we hit congestion
    - This procedure is called **slow start** (even though it's exponentially fast!)



Q: Why not additive decrease?
A: It does not converge to fairness; from a congested point, (x',y'), reducing each by 1 worsens fairness and takes us away from the "ideal" outcome.

Weaknesses of TCP: TCP is a massive success, but does have some drawbacks:

- Drives network to congestion in its probing for the correct window. If routers have too much buffering, causes long delays. We could make routers have the right sizes, and also incorporate other techniques (next lecture), but a more robust approach would be to work well no matter what the routers are doing.
- Assumes packet loss is due to congestion, rather than errors (which is more common in wireless). Reducing the window size in the face of stochastic, non-congestion losses is not a good plan.
- In high bandwidth, low delay situations, like in datacenters, TCP doesn't perform well. Similarly, in high bandwidth-delay product situations, takes a long time to converge.
- The TCP algorithm has a bias against long RTTs: throughput inversely proportionally to RTT (consider when sending packets really far away vs really close).
- Assumes cooperating sources, which isn't always a good assumption.