Lecture 7: Congestion control

Anirudh Sivaraman

2020/09/30

Last lecture, we looked at the phenonmenon of congestion collapse, i.e., what happens when you pick a very large window size in the sliding window protocol. The problem of congestion collapse on the Internet was first documented by John Nagle in 1984 [2]. It was also observed by Van Jacobson in 1986 [8]. In both cases, the congestion collapse was the result of a large number of packets being prematurely retransmitted. The network was still doing work and heavily utilized, but not all of it was contributing to useful transport-layer throughput because the receiver was receiving duplicate copies of the same packet. Figure 3 of Van Jacobson's paper [8] illustrates this problem beautifully. Quoting the caption of Figure 3 from that paper: "The dashed line shows the 20 KBps bandwidth available for this connection. Only 35% of this bandwidth was used; the rest was wasted on retransmits."

The solution to the Internet's congestion collapse problem was a *congestion-control algorithm*. This algorithm was first implemented on end hosts by Van Jacobson in the BSD Operating System around 1986-87 and published in 1988 [8]. This algorithm is today called AIMD (Additive Increase Multiple Decrease); the reason for this will become apparent soon. AIMD allows a sliding window sender to adaptively find a window size equal to the BDP of the network.[1] This is unlike what we have done so far, where we hardcoded a *fixed* window size into the sliding window protocol.[2]

AIMD was first developed by Ramakrishnan and Jain at Digital Equipment Corporation (DEC) in an algorithm that later came to be called DECbit [9]. However, Ramakrishnan and Jain's evaluation was confined to simulated networks and required changes to the routers—something that was quite antithetic to the Internet's ethic of minimalism in the routers. Van Jacobson then adapted the same algorithm for use in the Internet without modifying the routers and evaluated it on a real network. DECbit was eventually adopted and standardized as the Explicit Congestion Notification (ECN) mechanism in routers in 2001. But it took a while, unlike Van Jacobson's approach, which was implemented in every Internet end host by the early 1990s. This is another example of keeping things simple on the Internet: by minimizing the number of entities that need to be modified, Van Jacobson's approach was more readily adopted.

We'll discuss AIMD in more detail now. There are two main parts to this AIMD algorithm:

1. *Slow Start*, which allows a sliding window sender to discover the right value of the window size starting from an extremely conservative initial window.
2. *Congestion Avoidance*, which allows a sliding window sender that has reached the right window size to remain close to the right value as other senders arrive and depart from the network.

As a rough analogy, Slow Start is similar to accelerating to get to the right speed on the highway, while Congestion Avoidance is similar to maintaining this speed.

1 Slow Start

Slow Start allows a sliding window sender to discover the "right" value of the window size given the link's capacity, its minimum round-trip time, and the number of other senders on the network. It works in a manner similar to binary search.

[1]When n flows share a bottleneck, the algorithm finds a window size approximately equal to $\frac{BDP}{n}$

[2]Prior to AIMD, TCP's behavior was essentially similar to our sliding window protocol. It used a fixed window [1]. This window differed between different Operating System implementations of TCP.

The sender starts off with an initial window of packets, and on every ACK, increases the window by 1 packet to reflect the fact that the previous window size worked (i.e., you received an ACK) and it is OK to increase the window a little bit further. That's the essence of the algorithm. Let's see how it plays out over time.

At time 0, the sender sends out an initial window worth of packets. Let's say that this initial window is 1 for simplicity. An RTT later, the sender receives an ACK for the first packet. This ACK gives the sender permission to do two things:

1. As per the sliding window (and Stop-And-Wait) protocols, it allows the sender to release one more packet because one packet has been acked and the sender is now permitted to have one more unacked packet.
2. As per Slow Start, the sender can increase the window itself by 1. Hence, the window is now 1+1=2, and the sender is permitted to have two unacked packets. This allows the sender to release an additional packet, over and above the packet from the sliding window protocol above.

Hence, at the end of an RTT, the sender releases two packets. These two packets are acked another RTT later. These two packets in turn trigger two new packets—one each for sliding window and Slow Start. This means a total of 4 packets are sent out at time $2RTT$. You can probably see where I am going with this.

At the end of $3RTT$, 8 packets are sent out. At the end of $4RTT$, 16 packets are sent out, and so on. In other words, even though the window only increases by 1 on every ACK, it doubles every RTT. In other words, the window grows exponentially with time when the sender is in Slow Start. So, if W is the "right" value of the window, within $log_2(W)$ RTTs, the window has reached its right value of W. At this point, you might be wondering why it is called Slow Start, when the window growth is exponential in time. Historically, this algorithm was called Slow Start because it was slow relative to what TCP did before AIMD, which was to dump an entire large window of packets (i.e., W) onto the network at time 0 without ramping up to it.

How does the window ever stop doubling? At some point, the bottleneck link for a particular sliding window flow drops a packet from this flow. This is an example of a *congestion signal*: a sign that something is going wrong in the network and that corrective action needs to be taken. The sender can detect this packet loss in one of two ways. In the DECbit scheme, developed at DEC, the router would set a bit in a packet's header indicating that the average queue size had crossed a threshold. In Van Jacobson's algorithm, the sender would detect a packet loss using a retransmission timeout or when the sender receives acknowledgements for later packets. Becaue Jacobson's algorithm needed no changes to the router, it was much easier to adopt.

Once the sender has detected a loss, it cuts its window down to half of the window size at the time of the packet loss. After this, it enters a mode called congestion avoidance, where it makes much more gentle changes to the window relative to Slow Start.

2 Congestion Avoidance

In Congestion Avoidance, the sender continuously tries to (1) increase its window in case spare link capacity has opened up because an old sender left the network and (2) decrease its window in case spare link capacity went down because a new sender just joined the network. We'll now describe these increase and decrease rules.

In Congestion Avoidance, the sender increases its window by 1 every RTT, as opposed to 1 every ACK in Slow Start. Effectively, in Congestion Avoidance, this means the window increases *additively* by a constant amount. On the other hand, in Slow Start, the window increases *multiplicatively*: it doubles every RTT. In practice, to ensure the window increase is gradual, the window is increased by $\frac{1}{W}$ every time it receives an ACK, where W is the current value of the window. This translates into the same 1 unit increase every RTT.

In Congestion Avoidance, the sender decreases its window every time it detects a loss, either through a timeout or by the receipt of an acknowledgement for a later packet. It decreases its window in the same way as Slow Start, by halving the window. In other words, the decrease is multiplicative.

3 Why AIMD? (Won't be tested)

The name AIMD should make sense now. It captures the fact that the window size increases additivity, but decreases multiplicatively. Why pick this combination? You could imagine several other possibilities: Additive

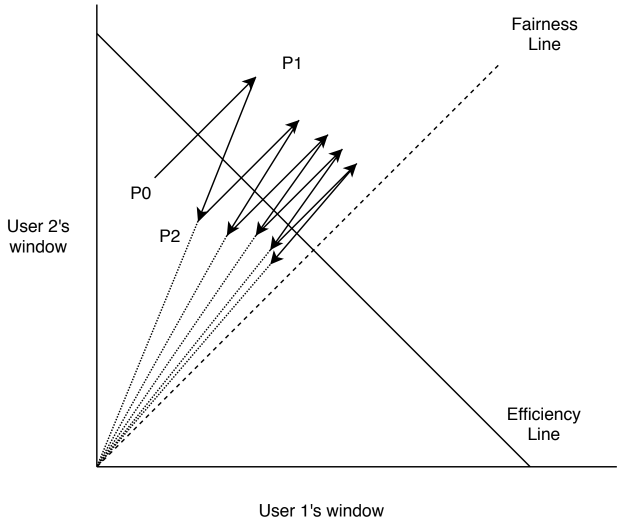

Figure 1: AIMD algorithm converges to a fair allocation of network resources

Increase Additive Decrease (AIAD), Multiplicative Increase Multiplicative Decrease (MIMD), and Multiplicative Increase Additive Decrease (MIAD). The short answer is that you want to be ginger when increasing so that you don't overwhelm the network, but you want to back off really quickly at the first sign of trouble. In other words, it is easier to drive the network into overload and hence we should be careful about increasing. But, it's not terrible if we are going to be underutilizing the network a little bit for a short period of time. Hence, we should back off quickly.

Now for the longer answer. Chiu and Jain provide both an algebraic proof and a geometric argument for why it should be AIMD in their paper [6]. I'll summarize the geometric argument here. Let's say you have two senders, whose windows are $x1$ and $x2$. We can represent these two rates by points on a 2D plot.

For an *efficient* allocation of the network's resources, the sum of $x1$ and $x2$ must be the same as the BDP of the network. This is the straight line with the equation $x1 + x2 = BDP$. We'll call this the efficiency line. For a *fair* allocation of the network's resources, $x1$ must equal $x2$. This is the straight line with the equation $x1 = x2$. We'll call this the fairness line. The intersection of the fairness and efficiency lines represents a fair *and* efficient allocation, i.e., $x1 = x2 = \frac{BDP}{2}$. A good algorithm will drive the network to this point no matter what the initial values of $x1$ and $x2$ are.

Let's look at each of the four policies in this regard. We'll assume synchronized instantaneous feedback, i.e., both senders increase/decrease their windows simultaneously and there is no delay between when the packets are dropped and when the senders know about the packets being dropped. Also, because all senders run the same protocol, the additive increment/decrement in the same for both senders, and the multiplicative increase/decrease factor is the same for both senders.

For MIMD, the fairness does not change no matter how the two senders increase or decrease their windows. In other words, an unfair allocation will persistently remain unfair, unless both senders decrease their allocations to zero, which is far from efficient. For AIAD, the network's trajectory will oscillate along a line perpendicular to the efficiency line. It can converge to an efficient allocation, without ever getting close to the fair and efficient allocation. For MIAD, the network will progressively get further and further away from the fair and efficient allocation, because every multiplicative increase will take the network further away from the desired position and the additive decrease cannot do enough to bring it back.

This leaves us with AIMD. For AIMD, every additive increase multiplicative decrease cycle improves fairness just a little bit more, while every additive increase takes the network closer to the efficiency line (Refer to Figure 1). Eventually, the network oscillates around the fair and efficient allocation. This is similar to how AIAD oscillates around a line perpendicular to the efficiency line. Except now, the MD component drives the network close to a fair allocation as well.

4 Congestion control today (or) what's happened since 1988?

Packet loss is just one very specific example of a congestion signal. Over the years, it was discovered that packet loss was not particularly robust as a signal of congestion because it depended on how big the router's buffer was. Further, between 1988 and now, it has become easy to build routers with larger packet buffers as memory costs have gone down. As a result, loss notifications might come too late, i.e, only after the buffer has built up to a point of extremely large queueing delays [5]. This could lead to a latency-based congestion collapse similar to the first example in the previous lecture.

There has been a rich literature of congestion control algorithms since Jacobson's paper. Developing good congestion control algorithms still remains a widely researched area of networking with considerable practical impact. It's probably not a stretch to say that this is one of the central problems of networking. It has remained important despite changes both in the underlying link technology that support the Internet and the applications running on the top of the Internet.

I will summarize some developments in congestion control over the years. To keep this short, I will focus on developments that are known to have been deployed in production networks for some period of time. macOS and Linux use a congestion control algorithm called TCP Cubic [7], published in 2008, which uses a different increase rule from AIMD. Windows uses a congestion control algorithm called Compound TCP, published in 2005 [10], which uses both packet delay and loss as signals of congestion. In 2010, researchers at Stanford and Microsoft developed a new congestion controller called DCTCP (Data Center TCP), which uses the ECN mechanism to perform congestion control in a data center [3]. DCTCP (or some variant of it) is developed in the private networks (also called datacenters) of many large companies today. [3] More recently, in 2016, Google developed a new a congestion control algorithm for wide-area, transcontinental Internet paths called BBR [4], which is in use on the Google Cloud Platform and is being experimented with for other Google services.

The major differences between different congestion controllers lies in what congestion signal they use (e.g., loss, delay, loss and delay, gradient of delay, etc.), who generates this congestion signal (e.g., router, receiver, or sender), what the senders do in response to these congestion signals (e.g., the multiplicative decrease rule), and how senders increase their window size (e.g., additive increase or slow start).

References

[1] Transmission Control Protocol, DARPA Internet Program, Protocol Specification. `https://tools.ietf.org/html/rfc793`, 1981.

[2] Congestion Control in IP/TCP Internetworks. `https://tools.ietf.org/html/rfc896`, 1984.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5), October 2016.

[5] Vint Cerf, Van Jacobson, Nick Weaver, and Jim Gettys. BufferBloat: What's Wrong with the Internet? *ACM Queue*, 2011.

[6] Dah-Ming Chiu and Raj Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 1989.

[7] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Operating Systems Review*, 2008.

[8] V. Jacobson. Congestion Avoidance and Control. SIGCOMM, 1988.

[3]DCTCP shares many similarities with DECbit [9], most notably the use of router support to indicate congestion. DECbit had trouble finding deployment in the 1980s due to the lack of router support for indicating congestion. However, by 2010, many routers supported the ECN mechanism, which was inspired by DECbit. DCTCP leverages this ECN mechanism.

[9] K. K. Ramakrishnan and Raj Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer. SIGCOMM, 1988.

[10] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. Technical report, July 2005.