Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.829 Fall 2002 **Lecture 9: Router-Assisted Congestion Control** October 3 2002

Overview. The gateway side of congestion control: explicit congestion notification and active queue management. RED as an example. Ensuring/enforcing cooperation of end-systems.

Note: These notes do not discuss XCP.

1 Introduction

In the previous lecture, we studied the principles of end-to-end congestion control and the practice of one of them (TCP). While being an extremely successful model for handling congestion problems, specifically in making sure that the response to congestion is sound (reduction in sending speed) and that the probing for bandwidth is carefdully done (slow start and additive-increase), end-to-end congestion control has some limitations.

1. Routers are the place where congestion occurs, so it would make sense to try and do smarter things at the routers with regard to congestion control. However, the trade-off here is making sure that what's running in the routers isn't overly complex or slow.
2. Purely end-to-end congestion control cannot enforce *isolation* between flows. If we want the rate of transmissions of flows to not affect each other, a purely end-to-end control strategy won't suffice in general. Depending on the economic model for network service, isolation may be a desired goal.
3. Whereas the viability of end-to-end congestion control relies on cooperating sources, it isn't always a good assumption in today's Internet.

What can routers do to overcome these limitations of end-to-end congestion control?

1. **Congestion signaling.** Routers can signal congestion when it occurs or is about to occur, to the sources in the network. This signaling can take on many forms: packet drops, explicit markings on packets, and explicit messages sent to sources. Some are more practical than others.
2. **Buffer management.** Routers have queues, primarily to absorb and accommodate bursts of packets. A router needs to decide *when* to signal congestion, and if it decides to, *which* packet to mark or drop to signal congestion. These two decisions are made by the buffer managememt algorithm at the router. A good buffer management scheme provides good incentives for end-to-end congestion control to be implemented.
3. **Scheduling.** When packets from multiple connections share a router, which packet should be sent next? Scheduling deals with this decision.

One might think that if routers were to do all this, then end-to-end congestion control isn't needed. This is not true. To understand why, we need to understand what the dangers are of not doing any form of congestion control. As discussed in L3, the problem is the potential for *congestion collapse.*

In general, congestion collapse might take several forms and might occur for several reasons. The following is a taxonomy developed by Floyd.

1. *Classical collapse:* This is the congestion collapse caused by redundant retransmissions, such as when the TCP retransmission timeout ends up re-sending packets that aren't actually lost but onlu in queues en route.
2. *Fragmentation-based collapse:* Here, (large) datagrams are broken up into smaller fragments and sent. The loss of even a single fragment will cause the higher layer to transmit the entire datagram. (Think of running NFS over UDP across a congested network with a block size of 8 KB and an MTU of 1500 bytes! This is not hypothetical.)
3. *Control traffic-triggered collapse:* Some protocols use control packets separate from the packets used to send application payload. Examples include multicast group membership packets, routing updates, etc. If not designed carefully, control packets can start overwhelming a network.
4. *Collapse from stale packets:* Here, the network carries packets that are no longer wanted by user, because of long delays (and unnecessary retransmissions). Or because of ill-informed *push* data to users (e.g., applications like Pointcast when not configured properly.
5. *Undelivered packets:* This is the hardest form to tackle, and is caused by unresponsive traffic. Some applications also incrase their bandwidth use upon detecting loss, sometimes by increased FEC (FEC in itself isn't bad, it's the increased bandwidth use that is).

Consider the network shown in Figure 1. Here, there are two sources S_1 and S_2, and every router implements flow isolation using fair queueing. Despite this, end-to-end congestion adaptation is essential, for otherwise, the 1.5 Kbps middle link ends up sharing bandwidth across S_1 and S_2, but most of S_2's packets end up getting dropped on the downstream 128 Kbps link! The absence of end-to-end congestion control is wasteful indeed!

The rest of this lecture deals with congestion signaling strategies and buffer management. The next lecture concerns scheduling, specifically flow isolation using fair queueing.

2 Congestion signaling

There are three ways in which routers can signal congestion to the sources. The first, and most robust for wired networks, is by dropping packets. On wired networks, packet losses occur pretty much only because of congestion, and the right reaction is to reduce the speed of transmission (e.g., window reduction).

The second way is not to drop a packet when the need to signal congestion arises, but to *mark* it (well, actually mark a bit in its IP header). When the receiver receives this packet with a marked bit, it *echoes* it to the sender in its ACK. Upon receiving an ACK with a mark echoed on it, the sender does exactly the same thing it would've done on a packet loss-based congestion signal.

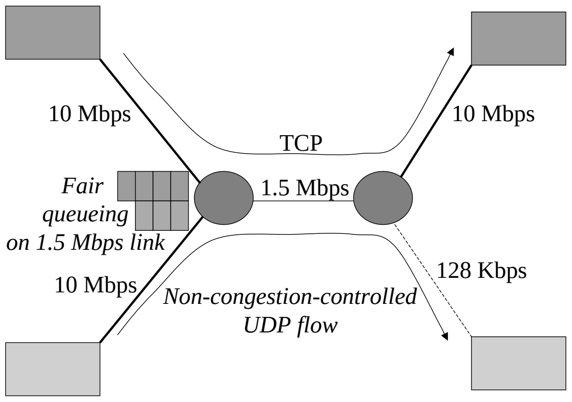
10 Mbps
TCP
10 Mbps
Fair queueing on 1.5 Mbps link
1.5 Mbps
128 Kbps
10 Mbps
Non-congestion-controlled UDP flow

Figure 1: End-to-end congestion control is essential even with ubiquitously deployed flow isolation mechanisms like fair queueing.

The third approach is to send an *explicit message* to the sender of the packet when the router decides that the network is congested (or congestion is incipient). This message is sometimes called a *source quench* and was once believed to be a good idea. It isn't used much today; one of the big problems with it is that it forces more packets to be sent precisely when congestion is occurring in the network!

2.1 Packet drops

On wired networks, packet drops almost always signify network congestion. The receiver in a reliable transport protocol indicates the loss of packets to the sender, which can perform the appropriate congestion control tasks. For protocols that don't require reliability, some form of feedback is still important in order to perform congestion control.

Not all schemes drop packets only when the router's queue is full. One can gain a great deal by making the drops *early*, so that the *average* queue size is still kept relatively small, even though the total queue size available is substantially larger, to accommodate bursts of packets. We will study this in Section 3.

2.2 Marking packets

Various proposals have been made for doing this. Historically, the DECBIT scheme (Ramakrishnan and Jain; RJ90 in the readings) from DEC combined an AIMD source with router support, where the routers would mark a bit in the packet header when the queue size (as estimated over a certain time interval, not the instantaneous queue size) exceeded some threshold.

The current standard way of signaling congestion using packet marking is called *Explicit Congestion Notification* (ECN). First worked out for TCP/IP by Floyd in 1994, it has recently been standardized. The idea here is simple:

1. The sender uses a special TCP option to tell the receiver it is capable of ECN, in the connection setup (SYN) packet.
2. If the receiver is capable of ECN, it responds in its SYN sent to the peer.
3. All subsequent packets on the connection have a bit set in the IP header that tell the routers that this packet belongs to a connection that understands, and will react to, ECN.
4. Each router may use its own policy to implement the ECN mechanism. For example, one might use RED, and mark the packet (by setting another bit in the IP header) when the average queue size exceeds some threshold, rather than dropping it. Of course, this marking is done *only if* the packet is found to be ECN-capable (i.e., the packet belongs to an ECN-capable connection or flow); otherwise, it is simply dropped. This is important since it allows both ECN and non-ECN flows to co-exist and compete fairly with each other.
5. Upon receiving any packet with ECN set on it, the receiver *echoes* this information in its ACK (or equivalent feedback) message to the sender.
6. When the sender receives an ACK or feedback message with ECN echoed, it takes appropriate congestion control measures, e.g., reduces its window. It also sets some information in the IP header that tells the receiver that the sender has in fact reacted to this echo.
7. To guard against possible ACK loss, the receiver sets ECN on *all* ACK messages until it receives a packet from the sender that indicates that in fact the sender has paid attention to this feedback. For reliable transport like TCP, this is sufficient since the receiver will eventually receive the packet from the sender that tells it that the sender has taken steps to deal with the information it received about congestion on the path.

There are several things to notice about the ECN design. First, an important goal is backward compatibility with existing TCPs, both in terms of the messaging formats, and in terms of how ECN and non-ECN flows compete for bandwidth. Second, the ECN mechanism needs some attention to implementation details since bits in the IP header aren't freely available. Third, the ECN mechanism is susceptible to abuse by receivers, since the receiver can exchange information with the sender assuring it that it will echo ECN information, causing the sender to set the bits in the packet that tell the routers that the packet belongs to an ECN-capable connection. Then, when the receiver gets a packet with ECN set on it, it can silently *ignore* this and not echo it back to the sender! The end-result is that this malicious receiver can obtain a largely disproportionate amount of the bottleneck link's available bandwidth.

The solution proposed by Spring et al. [SEW01 in the optional readings] to this problem is elegant and simple: use *nonces* on packets such that the receiver can *prove* to the sender that it did in fact receive a packet without the "congestion-experienced" bits when that packet arrived with this information set. In principle, this can be done by the source setting a random number in the packet header. When a router sees an ECN-capable connection and is about to set the "congestion-experienced" bits, it also *clears the nonce* to zero. The receiver, now, cannot lie that it received a packet without notification about congestion, since then it would have to echo the correct nonce!

The paper asserts that *one-bit nonces* suffice for this. It isn't hard to see why, since the probability of guessing a 1-bit random nonce correctly is 0.5. So, to be successful at lying k times falls off as $\frac{1}{2^k}$, which is rapid. Furthermore, upon discovering a lying receiver, the sender (if it wishes) can treat it punitively, by giving it much less than its fair share.

The final trick to completing the scheme is to make sure it works when not every packet is acknowledged. This is handled by making the nonces *cumulative* (i.e., the sum of all nonces sent so far, in $(0, 1)$ arithmetic). Furthermore, when the receiver cannot reconstruct the nonce (because it was cleared by a router on congestion), the sender needs to resynchronize its notion of the cumulative value with the receivers, and continue from there.

3 Buffer management, active queue management, and RED

Buffer management schemes at routers decide *when to drop a packet* and *which packet to drop*. The simplest scheme is *drop-tail*, where packets are accommodated until the queue is full, after which all incoming packets are dropped until queue space is again available.

This has several drawbacks: it doesn't signal congestion early enough, it causes packets to be dropped in bursts, and it doesn't keep average queue sizes small. This means that flows experience higher delays than might be reasonable.

It's worth noting that most of the Internet today still runs drop-tail gateways. The performance of drop-tail for TCP traffic under high degrees of statistical multiplexing isn't particularly well-understood.

4 RED: Random Early Detection

RED is an *active queue management* scheme, which explicitly tries to monitor and contain the average queue size to be small—small enough for delays to be small, but large enough for bottleneck link utilization to be high when bursty traffic causes a temporary lull in offered load (e.g., when a set of TCP sources cut down their sending speeds).

The key idea in RED is to use randomization to avoid synchronizations and biases that could occur in deterministic approaches, and to drop or mark packets earlier than when the queue is full.

The approach is rather simple: If the average queue size, q_a is between min_{th} and max_{th} the packet is *marked* or *dropped* packet with some probability, p_a. If $q_a > max_{th}$, then the packet is (always) dropped. If $q_a < min_{th}$, then the packet is forwarded through.

Two key issues in RED are: (1) how to compute q_a and (2) how to compute p_a.

4.1 Computing q_a

q_a is computed using a standard EWMA filter:

$$q_a = (1 - w_q)q_a + w_q q$$

Here w_q is time constant of low pass filter. If w_q is too large, it implies a rapid response to *transient* congestion. If it is too low, it implies sluggish behavior. The paper shows a back-of-the-envelop for setting this: obtain an upper bound based on how much burstiness you want to allow, and obtain a lower bound based on how much memory you want to maintain. It isn't clear to me how well this works in practice.

The paper uses $w_q = 0.002$. Also, the recommendation is for $max_{th} \geq 2min_{th}$. The best place to find out how these parameters should be set is `http://www.aciri.org/floyd/red.html`.

4.2 Computing p_a

Initially, compute $p_b = max_p \times \frac{q_a - min_{th}}{max_{th} - min_{th}}$. This is standard linear interpolation.

There are two ways in which p_a, the actual drop probability, can be derived from p_b.

1. Method #1: $p_a = p_b$. This leads to a geometric random variable (r.v.) for inter-arrivals between markings (or drops). Consider the r.v. X = number of packets between successive marked packets. Then $P(X = n) = (1 - p_b)^{n-1} p_b$. This is a geometric distribution.

   Unfortunately, this doesn't mark packets at fairly regular intervals. $E[X] = 1/p_b$, but because packets get marked in bunches, more synchronization than is good could occur. $Var[X] = 1/p_b^2 - 1/p_b$, which gets big for small p_b.

2. Method #2: Uniform r.v. for inter-arrivals between markings or drops. Let X be uniform in $\{1, 2, \ldots, 1/p_b - 1\}$. This has the nice property that we want, and is achieved if we set $p_a = p_b/(1 - count \times p_b)$, where *count* is the number of unmarked packets since last marked packet. $E[X] = 1/(2p_b)$.

The final point is about the choice of max_p. One argument is that it should never have to be large, since after $q_a > max_{th}$, all packets are dropped anyway. However, a small value of max_p (like 0.02) makes the behavior of p_b rather abrupt and discontinuous when $q_a \approx max_{th}$. A "gentle" variant of RED has been proposed that makes it less discontinuous, or indeed continuous.

Several variants of RED and alternatives to it have been proposed in the last few years. A search on Netbib (see the "Useful Links" page from the class Web page) or the Web will give you some pointers. Some examples include Gentle RED (where the transition from probabilistic to always drop is not so sudden), ARED, SRED, AVQ, BLUE, CHOKE, etc.

4.3 Encouraging end-to-end congestion control

In principle, active queue management schemes, and RED in particular, can also handle misbehaving or uncooperative sources, by penalizing them after detecting misbhavior ("penalty box"). In theory, this can be done (for example) by RED keeping an "drop history" of connections that incurred RED drops (or marks) and periodically scanning this history to see if there are packets from any particular connections that dominate this history.

If this sort of mechanism were implementable, then it would form the basis for the right incentive structure to *encourage* the use of end-to-end congestion control mechanisms.

With RED, in the long run, a connection's share of marked (or probabilistically dropped) packets is directly proportional to its share of bandwidth. This is because RED equalizes drop rates (the ratio of number of dropped packets to number of sent packets) across competing flows. (Note however that RED doesn't provide flow isolation and isn't fair in the fair queueing sense of max-min fairness.)

It can be shown (see, e.g, FF99 in the optional readings) that for a fixed q_{avg}, n packet drops, and a fraction b of arriving bandwidth, P(flow receives $> 2 \times$ its "fair share" of drops) $\leq e^{-2nb^2}$.

So, over a sufficiently large number of drops (and a drop rate that's not too large), the RED packet drop history should be a good estimator of the *long-term* arrival rate of a flow.

One important point to note is that we haven't actually specified what characterizes an "ill-behaved" flow, the type of flow we're trying to penalize. There are several possible notions of this:

1. Flows that aren't *TCP-friendly* (see L8 for what this means). These are more aggressive than the throughput $\propto \frac{1}{\sqrt{p}}$ relationship for conformant TCPs.

2. Flows that are *unresponsive*. These don't reduce their rate in proportion to increases in loss rate.

3. Flows that are using *disproportionate bandwidth*. These are both responsive and TCP-friendly, but using more than their fair share (e.g., "significantly" more than $1/n$ when there are n flows in all).

Depending on the goals of the network provider, tests for some or all of these will be useful (but these are usually difficult to devise). When a misbehaving flow is detected, this flow's packets would be placed in a "penalty box" on the forwarding path, and not serviced in a timely manner. This provides a *disincentive* against being unresponsive to congestion.