Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033, Spring 2000 **How to Efficiently Share a Network** March 13, 2000

---

1 Introduction

In past lectures, we have worked our way up from the link layer via the network layer to the end-to-end layer, discussing issues such as link-layer framing, error detection, addressing, routing, packet forwarding, loss detection, retransmissions, end-to-end flow control, and connection management (connection establishment and teardown).

We now turn our attention to a challenging problem that is fundamental to networking—*resource management.* This is a fundamental issue because the notion of *sharing* is central to cost-effective and scalable network design.

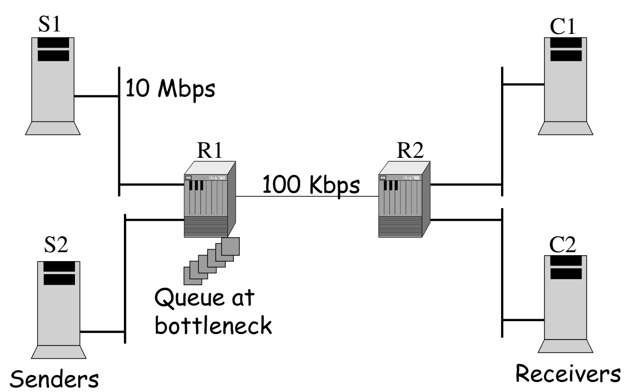

Figure 1: A simple network topology showing how sharing causes resource contention. Many connections between senders and receivers contend for the bottleneck 100 Kbps link.

The first question one may ask at this stage is: "What resources are being shared that need to be managed?" To answer this question, let's think about some of the "interesting" properties of best-effort networks from the previous lectures. At any router, each packet is processed and forwarded along one of several possible output links, at a transmission rate determined by the rated bandwidth of that link. In the meantime, more packets, which in general could belong to any other flow[1], could arrive at the router (recall that this is one of the key consequences of asynchronous multiplexing

[1] Or connection. A flow is just a generalization of a connection; the distinction between the two is not important at this stage.

in data networks). What happens to these packets? The router tries to accomodate them in its queues to eventually process them, but if there isn't enough space in its queue, some of them may be dropped. Thus, packet queues build up when the link is busy and demand for the link outstrips the available link bandwidth, and packets may be dropped when the queues get full. Figure 1 shows a simple network with connections between senders S_i and receivers C_i via a 100 Kbps bottleneck link[2].

The above discussion makes it clear that there are two resources that flows contend for in a network:

1. **Bandwidth.** The network has to decide how to apportion bandwidth between different flows. Network routers may also decide to prioritize certain types of packets (e.g., latency-sensitive audio or `telnet` packets) over others (e.g., electronic mail)[3].
2. **Queue space.** When the router decides that a packet has to be dropped because it is running out of queue space, which packet should it drop? The arriving one? The earliest one? A random one? And when should it decide to drop packets: only when the queue is full, or sooner than that? Waiting too long to before dropping packets only serves to increase packet delays, and it may be advantageous to occasionally drop packets even when the queue isn't full to inform the senders that the network is starting to get overloaded.

What happens if we don't manage network resources well? For one thing, the available bandwidth might end up being greatly under-utilized even when there is demand for it, causing economic heartburn. Most often, however, network designers end up provisioning for a certain amount of "expected" *offered load,* and then have to deal with overload, or *congestion.*

1.1 Understanding congestion

A network link is said to be *congested* if contention for it causes queues to build up and for packets to start getting dropped. At such a time, demand for link bandwidth (and as a result, queue space), outstrips what is available, leading to congestion. For this reason, many network resource management problems are also called *congestion control* or *congestion management* problems, since managing congestion efficiently implies an efficient management of the above network resources.

There are many examples of how overload causes congestion. For example, a possible way to achieve high throughput in a network might be to have all the sources blast packets as fast as they can, so that bottleneck network links run at close to 100% utilization. While this seems reasonable, a little thought shows that this approach is self-defeating. All it really accomplishes are long packet queues and resultant end-to-end delays, as well as increased packet loss rates, which for a reliable end-to-end layer would cause a large number of retransmissions. These retransmissions end up taking up more bandwidth in turn. This is therefore quite the wrong thing to do to obtain good network throughput. This example also demonstrates the fundamental tension between achieving high link utilizations on the one hand (by transmitting data rapidly), and low delays and loss rates on the other (which increase if data is sent too rapidly from senders).

This notion of how increasing offered load to achieve high utilization is at odds with packet losses and delays is illustrated in Figure 2. This figure shows a schematic view of throughput as a function

[2]We use Mbps for Megabits per second, Kbps for Kilobits per second, and bps for bits per second

[3]The careful reader will recall the type-of-service (TOS) field in the IP header from the layering lecture; routers sometimes look at this field to decide how to prioritize packets

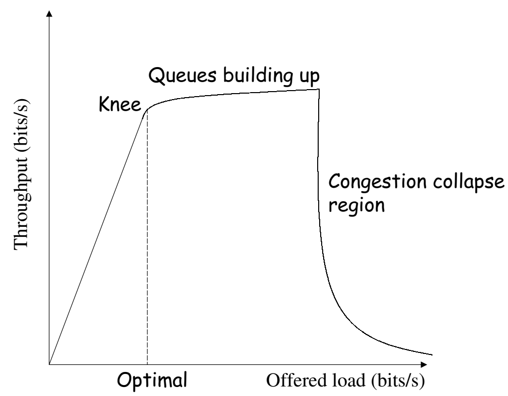

Figure 2: Schematic view of throughput vs. offered load, showing the optimal operating point and congestion collapse region.

of the offered load on the network. Initially, at low levels of offered load, throughput is roughly proportional to offered load, because the network is under-utilized. Then, throughput plateaus at a value equal to the bottleneck link bandwidth because packets start getting queued.

After a while, when offered load is increased even further, the throughput shows a "cliff" effect, starting to decrease and eventually go to zero, e.g., because every lost packet is being retransmitted, using up additional bandwidth! Now, all the competing flows are sending data rapidly, but no user is getting any useful work done (low network throughput). The network has just suffered *congestion collapse*—a situation where observed throughput *decreases* in response to an increase in offered load. While seemingly impossible, this catastrophic result is quite real and occurs in a number of scenarios; for example, the mid-1980s did see this catastrophic event several times, with large portions of the then Internet coming to a standstill.

Congestion collapse is an example of an *emergent property,* one that shows up rather suddenly in certain situations when the network size and load exceeds a certain amount. Congestion control protocols attempt to reduce or eliminate the occurrence of such nasty situations. In particular, the goal for a good network resource management algorithm would be to operate near the left knee of the curve in Figure 2, by modulating the sources to provide the appropriate optimal offered load.

1.2 Goals of network resource management

At first sight, it might seem that network resource management problems are not that hard to solve. A particularly simple approach would be for a centralized network manager to arbitrate and decide on how link bandwidths and queue buffers are apportioned between different flows, and perform "hard-edged admission control" (Section A of Chapter 4 describes what this term means) when new flows are started. Unfortunately, this "solution" turns out to be impractical for networks larger than only a small number of nodes.

A second approach might be to throw money at the problem and hope it will go away. In particular, we could decide to provision lots of queue space in our network routers, so the worst that happens upon overload is that packets get queued and are rarely discarded. Unfortunately, all that this does is to increase the delays incurred by new packets and does not improve the throughput or efficiency of the system at all. This is a little bit like long lines at an amusement park, where people are waiting to go on a popular roller coaster ride; adding more and more space for people to wait in line doesn't improve system throughput at all! The real purpose of queues is primarily to absorb bursts of data that arrive from senders, without causing long delays. We need queues, but we should be careful how many packets we queue at each router; persistent queueing of packets is not a good thing.

What are the goals of network resource management in large, heterogeneous networks like the Internet? There are several desirable properties that any good solution must possess, which make the problem interesting and challenging:

- **Efficiency.** We would like our solutions to lead to high network utilization. What this translates to is high end-to-end application throughputs and low end-to-end delays for all the competing flows.
- **Fairness.** The next goal is fairness— we want fair allocations of resources to competing flows. After all, it is possible to achieve high efficiency by starving all flows but one, obviously an undesirable end result.

  It turns out that coming up with the right metric for fairness is tricky, and there are many possible definitions. One possible way of measuring fairness is via a *fairness index*, which is a function of the mean and variance of the throughputs achieved by different streams. If $x_1, x_2, \ldots, x_n$ are the throughputs of n competing flows, the fairness index f is defined as

  $$f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}.$$

  Clearly, $1/n \leq f \leq 1$, with a smaller value of f signifying a larger degree of unfairness in allocations.
- **Distributed operation.** For even moderately sized networks, we can't assume the existence of a centralized oracle deciding what different flows should do.
- **Convergence.** Because of the inherently distributed nature of the system, each node (router or end-host) only has incomplete knowledge of the traffic in the network, with which it must act. An important goal of a practical algorithm is that it must not display large oscillatory behavior, and must try to converge towards high efficiency and fairness.

2 Taxonomy of Solutions

Before discussing technical approaches to solving this problem, let's ask ourselves if just spending more money and adding more bandwidth to network links will eliminate the problem altogether? It turns out that this doesn't seem plausible, based on past experience and predicted future traffic patterns. First, there is an increasingly wide realm of bandwidth ranges that the network needs to handle efficiently, because of the large number of network access technologies that are used. Second, the number of users and flows is increasing very rapidly. And third, various studies have shown the *per-user* bandwidth needs and consumption have been increasing with time, at the rate of at least 10-15% per year. Thus, increasing link bandwidths is essential to support increased populations, but will not eliminate the problems (in fact, these problems have become worse with time in the past few years).

One of the important factors that makes network resource management challenging is its cross-cutting nature—a good solution involves different layers of a protocol stack. We distinguish four classes of solutions to the resource management problem: *end-to-end flow control, router-based congestion control, end-to-end congestion control,* and *pricing-based resource management.* A comprehensive practical solution often composes these different classes together in interesting ways.

Using end-to-end flow control, a receiver attempts to ensure that the sender does not send data too rapidly to it and overrun its buffers. Router-based congestion control schemes implement different flow scheduling and queue space management algorithms at congested routers to alleviate congestion and improve fairness. End-to-end congestion control decide how data senders (at the end-to-end layer) react when congestion occurs, and how they behave in the absence of congestion. Finally, pricing relies on economic principles to manage network load, where network providers charge for network access and attempt to alleviate congestion by adjusting the price charged to users as a function of user demand. All of these are still topics of much active research, and there are several trade-offs to consider in practice.

3 Flow Control

This section describes the rationale and details of end-to-end flow control protocols. Consider an extremely simple network, a sender connected to a receiver over a fast 100 Mbps link connected via a fast router. Suppose the sender is a fast 450 MHz PC, while the receiver is a slower 100 MHz machine. Assume that any sent data arrives in an 8 KByte receiver buffer for the receiver's end-to-end layer to process. Clearly, if the sender sent data faster than the receiver could process, its buffer would overflow and packets will be lost. The receiver and sender need to negotiate how much data the sender is allowed to send at any point in time, and the sender must not violate this agreement. This agreement is also called *flow control,* and is the result of negotiation between the end-to-end layers at the receiver and sender.

Most protocols implement flow control via windows. Here, every acknowledgment packet (ack) to the sender also tells the sender how much the sender can send at this point without overflowing receiver buffers. The sender can now be reasonably certain that as long as this agreement is not violated, no packets will be dropped because of overflow at the receiver.

Ensuring flow control is relatively easy—just have a simple agreement between sender and receiver on how much each window can be. However, this does not ensure that packets will not be lost anywhere else, because the network could easily be the bottleneck and become congested. For example, this can easily happen for the topology shown in Figure 1, especially when several flows are sharing the 100 Kbps link. To function well, the network needs a *congestion control* mechanism, which is significantly harder because neither the sender nor the receiver has precise knowledge about where the bottleneck might be or what the bottleneck bandwidth is.

Thus, flow control is an end-to-end protocol to avoid overwhelming the receiver application and is *different* from congestion control, which uses a combination of techniques to avoid overwhelming the routers and links in the network. Flow control is usually achieved via a periodic window-based negotiation between sender and receiver. It is also significantly easier to solve than congestion control, as we will soon discover. Networks need *both* to function well.

4 Router-based Congestion Control

Router-based solutions are implemented in the switches and routers in the network. The simplest form of router support for congestion control is displayed in FIFO (first-in-first-out) queues at routers, which forward packets in the same order they were received, and drop arriving packets when the queue is full. In such a network, most of the burden for congestion control is pushed to the end-hosts, with the routers providing only the barest form of congestion feedback, via packet drops. The advantage of this minimalism is that the interior of the network (i.e., the routers) don't need to maintain any per-connection state and are extremely simple. The onus for dealing with congestion is pushed to the end-to-end layer, which uses packet losses as a signal that congestion has occurred, and takes appropriate action.

This is essentially the approach taken in most of the Internet today: most IP routers implement FIFO scheduling and *drop-tail* queue space management, dropping all packets from the "tail" of a full queue. This approach, of keeping the interior of the network largely stateless and simple, is in keeping with the fundamental design philosophy of the Internet, and is one of the reasons for the impressive scalability of the Internet with increasing numbers of users and flows.

In recent years, a number of sophisticated scheduling and queue management algorithms have been proposed for deploying in Internet routers, and more will be developed in the future. For example, routers could segregate the packets belonging to each flow into separate queues and process them in a "fair" way, using a technique similar to round-robin scheduling (a variant of this, developed some years ago, is called *fair queueing*). This approach is of course significantly more complex than simple FIFO queueing, because it requires per-flow state and processing to handle packets. Another approach called *random early drop (RED)* involves monitoring the average queue length and dropping packets "early" when this average exceeds a certain pre-configured threshold *even if* the queue is not full. This scheme allows senders to learn about router overload before the queue gets full and is useful in avoiding further congestion.

While the detailed study of these more sophisticated schemes is beyond the scope of 6.033, we note that even if these were widely deployed in the network, the end-to-end layer would still have to eventually deal with congestion and decide how to modulate its transmissions to avoid congestion to achieve high throughput and network efficiency. (It may be instructive for the reader to try and cast this in terms of an end-to-end argument.)

We now proceed to understanding how the end-to-end layer does congestion control. To make things specific, we focus on best-effort, FIFO networks.

5 End-to-end Congestion Control

5.1 Specifying the problem

End-to-end congestion control protocols address two fundamental questions:

1. How to react to congestion when it occurs, as signified by dropped packets?
2. How to determine the available capacity for a flow at any point in time?

When congestion occurs, all the flows that detect it must reduce their transmission rate. If they don't do so, then the network will remain in an unstable state, with queues continuing to build up. To move toward the optimal operating point of Figure 2, delays and loss rates must decrease, which can only happen if the congested flows react to congestion by decreasing their sending rates. A *decrease algorithm* determines how this is done by the end-to-end layer.

When packets are successfully being received and acknowledged by the receiver, the sender can deduce that the current rate is sustainable by the network. However, it does not know if the network can sustain still higher rates because available bandwidth varies with time as users start and depart, and so it needs to carefully probe for spare capacity to use. An *increase algorithm* determines how this is done by the end-to-end layer.

Before we can talk about the details of specific decrease and increases algorithms, we need to understand how the end-to-end layer at the sender modulates its transmission rate. We study a class of protocols called *window-based* protocols for doing this, since it is the most common way of doing it today (e.g., TCP uses this technique). Here, the sender maintains a window, also called the *congestion window,* or `cwnd`, that it uses to decide how many *unacknowledged* packets it can have in the network at any time. This window is different from the flow control window that we discussed before, which is negotiated by the sender and receiver to avoid overrunning the receiver's buffers. In contrast, `cwnd` is dynamically adjusted by the sender to try and track the state of congestion and available bandwidth in the network.

When the sender has packets to transmit, it sends out only as many as the *minimum* of these two windows permit, thereby ensuring that the number of outstanding, unacknowledged packets does not overrun the receiver nor (hopefully) overwhelm the network. In the rest of this discussion, we assume that the flow control window is larger than *cwnd*, or that the network rather than the receiver is the performance bottleneck. Because it takes roughly one round-trip time (`rtt`) to transmit a window's worth of packets, the equivalent transmission rate for such a window-based protocol is the ratio of `cwnd/rtt` packets per second.

5.2 Decrease and increase algorithms

We are now in a position to discuss the details of the decrease and increase algorithms. It is clear that there are an infinite number of ways to decrease and increase windows, and so we restrict ourselves to a simple, practical class of control algorithms called *linear* algorithms. Here, if w_i is the value of `cwnd` in the ith round-trip since the start of the connection,

$$w_{i+1} = aw_i + b$$

for real-numbered constants a and b. a and b are of course different for the decrease and increase cases, and ensure that w_{i+1} is smaller than or greater than w_i respectively.

It turns out that to ensure stable behavior and convergence toward fair allocations, a and b must be chosen for the decrease to be *multiplicative* and for the increase to be *additive.* That is, upon congestion

$$w_{i+1} = aw_i, 0 < a < 1,$$

and while probing for excess capacity,

$$w_{i+1} = w_i + b, 0 < b << w_{max},$$

where w_{max} is the largest possible window size (equal to the product of end-to-end delay and bottleneck bandwidth). The intuition behind this is that when the network is congested, the queue lengths start to increase exponentially with time, and the decrease must be multiplicative to ensure that this dissipates quickly. The increase algorithm must move gingerly and also try to move toward fair allocations of bandwidth, which is achieved by an additive algorithm.

For example, TCP sets $a = 0.5$ and $b = 1$. That is, when the sender detects congestion, it reduces its window to half the current value, and when an entire round-trip goes by and all the packets were successfully received (as indicated by all the ACKs arriving safely), it increases the window by 1 packet, carefully probing for a little more bandwidth to use.

In general, this technique for increasing and decreasing windows is also called *additive-increase/multiplicative-decrease (AIMD)* congestion control. Pseudo-code for this algorithm using TCP's constants is shown below.

```
// On detecting congestion via packet loss, do multiplicative-decrease
cwnd := cwnd / 2;
// When the current window is fully acknowledged (roughly one round-trip)
cwnd := cwnd + 1;
```

5.3 Conservation of packets

Good window-based congestion control protocols follow the *conservation of packets* principle, which says that for a connection "in equilibrium," the packet flow must be conservative (this is a term from fluid mechanics). A flow is in equilibrium if it is running with a full window of packets in transit, equal to the product of available bandwidth and delay (in the absence of any persistent queueing). Of course, the end-to-end layer knows neither the available bandwidth nor the non-queueing delay with any great accuracy, which is why it needs to *estimate* them via its increase algorithm and round-trip calculations. Note that it need not explicitly calculate either parameter for its congestion control, and can rely instead on the observation that in the absence of congestion, an entire window is acknowledged by the receiver in one round-trip duration. Then, the end-to-end congestion control protocol can ensure that packet conservation isn't violated by ensuring that a lost packet is not prematurely retransmitted (this would violate conservation because a packet presumed to be lost would still be in transit at the time of retransmission). An excessive number of premature retransmissions does in fact lead to congestion collapse, and this was the cause of the mid-1980s disaster.

TCP uses a widely applicable principle called *self-clocking* to send packets. Here, the sender uses ACKs as an *implicit* clock to regulate new packet transmissions. Clearly, the receiver can generate acks no faster than data packets can reach it through the network, which provides an elegant self-clocking mechanism by which the sender can strobe new packets into the network.

5.4 How TCP works: slow start, congestion avoidance and control

We now have sufficient background to use the above principles and apply them to a real protocol, TCP.

We start by observing that packet conservation applies to a connection in equilibrium, one which has reached its sweet-spot window size and is probing for more around it (and reacting to congestion using multiplicative decrease). But when a new connection starts, how does it know what this sweet spot is, and how does it get to it to get its ack-triggered "clock" going? TCP achieves this using a technique called *slow start.* Here, TCP sets its `cwnd` to 1 and sends one packet[4]. When an ACK arrives for it, it increases its window size by 1 and now sends 2 packets. Now, when an ACK arrives for *each* of these packets[5], *cwnd* increases by 1. That is, `cwnd` is now equal to 4. In the slow start phase, `cwnd` increases by 1 for each incoming ack. In this phase, TCP uses the ACK clock to not only clock packets out, but also to open up its window.

When does this process stop? Well, we would like to stop it when the window size reaches the equilibrium value (roughly the bandwidth-delay product), but of course we don't know what that is. So, we stop it at an estimate called the slow start threshold (`ssthresh`) or on packet loss (because that's a sign of congestion), after which we move into the additive increase phase described above. Of course, if we stop it due to congestion, `cwnd` multiplicatively decreases.

How fast does the window grow toward equilibrium in slow start? If the current `cwnd` is w_i, the sender transmits w_i packets within one `rtt`. If they all reach, w_i ACKs come back to the sender, *each of which* increases `cwnd` by 1. So, after all w_i packets are acknowledged (one `rtt` later), $w_{i+1} = 2w_i$. Thus, the window increase in slow start is *multiplicative,* increasing exponentially with time. Thus, slow start isn't really that slow, and reaches an equilibrium window size estimate W in $\log_2 W$ round-trips.

Once slow start completes, TCP uses the AIMD principle to adjust its `cwnd`. The only tricky part is in the increase pseudo-code, shown below.

```
// When an ACK arrives for a packet,
       cwnd := cwnd + 1/cwnd;
```

The $1/cwnd$ increment follows from our wanting to increase the congestion window by 1 after one round-trip. Because TCP wants the increase to be smooth rather than abrupt, we approximate it by increasing by $1/cwnd$ *when each packet is acknowledged.* When `cwnd` ACKs arrive, the effective increase is approximately 1, after about one round-trip. And `cwnd` ACKs would indeed arrive in the absence of congestion. This phase of TCP is also called the *additive phase* or *congestion avoidance* phase.

This also serves to point out the dual purpose of TCP ACKs—on the one hand, it is used to detect losses and perform retransmissions, and on the other, it is used to clock packet transmissions and adjust `cwnd`. Thus, any variability or disruption in the ACK stream will often lead to degraded end-to-end performance.

The final twist to TCP's congestion control algorithm occurs when a large number of losses occur in any window. Taking this as a sign of persistent congestion, TCP takes a long timeout before trying to resume transmission, in the hope that this congestion would clear up by then[6]. At this point, it has obviously lost its self-clock, and so goes back into slow start to resume; it sets its slow start threshold `ssthresh` now to one-half of `cwnd` at the time this persistent congestion occurred. Finally, for each failed retransmission, TCP performs *exponential backoff* for successive retransmissions by doubling the interval between attempts, similar to the *random exponential backoff* retry technique

[4]In reality, TCP maintains all its windows in terms of number of bytes, not packets. This only complicates the book-keeping and does not alter the principles or algorithms significantly.

[5]We assume that the receiver acknowledges every packet.

[6]It is actually untrue that the number of lost packets in a window is correlated with the degree of congestion in a drop-tail FIFO network. The real reason is that when a number of losses happen, TCP loses faith in its ACK clock and decides to start over.

used in the Ethernet[7]. One can show that with a large user population, you need at least an exponential backoff on congestion (or in the Ethernet case, collision) to be safe from congestion collapse.

Pseudo-code for TCP achieving its self-clock via slow start is shown below.

```
if (timeout) // connection in progress and many packets lost
        ssthresh := cwnd/2; // set ssthresh on timeout
else {       // connection start-up
        ssthresh := MAX_WIN; // max window size, 64 KBytes or more
}
cwnd := 1;
```

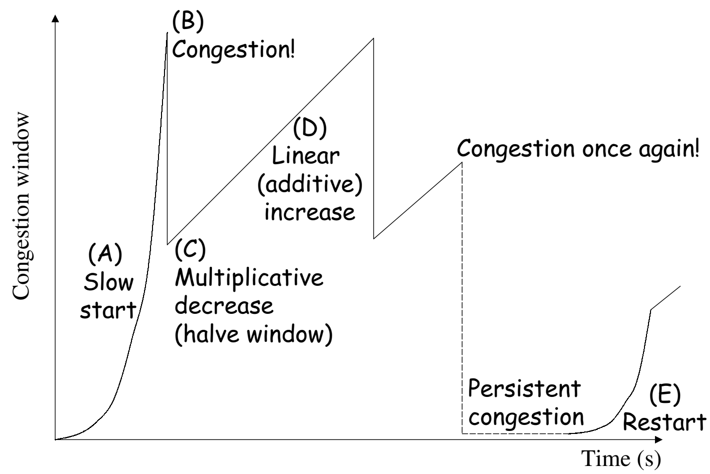

Figure 3: The time evolution of TCP's congestion window.

We are now in a position to sketch the evolution of the congestion window, `cwnd`, as a function of time for a TCP connection (Figure 3). Initially, it starts at 1 and in slow start doubles every `rtt`, (A) until congestion occurs (B). At this point, if the number of lost packets isn't large, the sender halves its transmission rate (C), and goes into the additive increase phase probing for more bandwidth (D). This sawtooth pattern continues, until several packets are lost in a window and TCP needs to restart its self-clocking machinery (E). The process now continues as if the connection just started, except now slow start terminates when `cwnd` reaches `ssthresh`.

We conclude this section by observing that the soundness of TCP's congestion management is to a large degree the main reason that the Internet hasn't crumbled in recent years despite the phenomenal growth in the number of users and traffic. Congestion management is still a hot research area, with additional challenges being posed by Web workloads (large numbers of short flows) and by real-time audio and video applications for which TCP is an inappropriate protocol because of the fully-reliable, byte-stream abstraction it provides.

[7]But without the randomization that the Ethernet uses to avoid synchronized retries by multiple stations.

6 Pricing

A radically different approach to network resource management would be to rely on pricing mechanisms and the associated economic incentives and disincentives to effect efficient and fair network use. Such approaches are based on the observation that bandwidth and queue space are simply economic commodities and that the economic principles of supply and demand ought to apply to them. Indeed, to some extent, telephone companies do control overload by variable-rate pricing (e.g., prices are cheaper during slack hours and more expensive during peak hours). For asynchronously multiplexed data networks, no one has really figured out how to do this and if it will work. In particular, while pricing is a promising mechanism to provision resources over sufficiently long time scales (e.g., weeks or months), relying exclusively on network service providers to deploy flexible pricing structures to keep up with the wide variation in traffic load over small time scales could be risky. Like other aspects of congestion management, this is also an area of active research.

7 Summary

This lecture discussed the important concepts and ideas in network resource management, an important and active area of work. A comprehensive solution to the problem cuts across the different layers of the protocol stack, and includes end-to-end flow control, router-based congestion control, end-to-end congestion control and pricing-based resource management using economic principles.

Acknowledgements

The presentation of this material was improved by comments from M. Frans Kaashoek, Jerome Saltzer, and Srinivasan Seshan.