6.02 - Circuit Switching, Packet Switching, Queues
Lecture #19
Katrina LaCurts, lacurts@mit.edu

Introduction to Networking

Networks as Graphs

We've done a lot so far in 6.02 with point-to-point communications, where one machine communicates with another machine.

Last week we talked about sharing a channel, but we still dealt with only point-to-point links. (You can model the ALOHA network as each node having a single link to the satellite, with the caveat that nodes can't transmit on their links at the same time. Alternatively, you could model the network as multiple nodes sharing the same link.)

What we would like to have now is many interconnected points.

We often model networks as graphs. Last week I used the term node, to reference the vertices in the graph. When nodes have an edge between them, it means there is a point-to-point link between the two. That link might be a wireless link, or it could be a physical wire between the two machines. Thinking of a physical wire between two connected nodes is a good internal representation for most of 6.02 (I'll let you know if it's ever not).

Sharing the Network

The first problem we're going to deal with in networking is how to share the network (similar to how we looked at how to share a single channel last week). What do I mean by sharing the network?

Consider the following: let's imagine that we have five endpoints that want to communicate. We want a way for each endpoint to be able to communicate with every other endpoint. So we could do this: we could just wire the network up to make a completely connected graph, where each machine has a physical wire to connect it to every other machine.

Do you see any problems with this? First of all, this scales terribly. If we have N nodes, we need O(N^2) wires. The Internet has billions of end points; this would necessitate quite a few wires. Perhaps even more of a concern is how we would physically get a wire between each machine and every other machine.

Clearly this is a terrible idea. So we do something else: we put switches in the middle of the network, and make sure there is a *path* between each node and every other node. (Note that now we have two types of nodes: end points and switches. End points are the things sending messages to one another; switches are the things sending the messages through the network.)

Switches are entities that sit in the middle of the network. They don't initiate communications themselves, but they're responsible for passing information through the network. I'll say more about them in a second.

This method of connecting machines is a much (*much*) better idea, but consider what happens when two connections are happening at once. They might need to share one of the links.

This is what I meant by "sharing the network". How should we do that? How do we divide the link among these connections?

In last week's lecture, you saw some ways to share a channel, which we could use to share individual links. Now that we are discussing larger networks than we were last week, we can put these ideas into more context, and we can also discuss some alternatives.

To discuss these alternatives, we need to take a closer look at what switches do. Switches solve three fundamental problems:

1. Forwarding - When data arrives at a switch, the switch needs to process it, and determine which outgoing link to send it on

2. Routing - Each switch needs to determine the topology of the network (i.e., how the nodes are connected; how to get data meant for machine X to machine X). This typically happens in the "background"; it's not an action that occurs on the arrival of each piece of data.

3. Resource allocation - Switches allocate resources to the different communications that are in progress. They have to decide how to share the capacity of the outgoing link among multiple connections that are using that link.

We'll talk more about problems 1 and 2 in later lectures.

In general, there are two approaches to sharing a network:
1. Circuit switching
2. Packet switching

How the switches behave in each of these approaches is a big part of how they differ.

==========
Approach 1: Circuit Switching
==========

Circuit switching is used in land-line phone networks, so a good way to think about this is to imagine what happens when you make a phone call from a land-line.

Here is the protocol for circuit switching:

1. Setup phase: Establish a circuit between the end points. I'll call them A and B for concreteness. To set up the circuit, the switches along the circuit configure some state in themselves.

2. Data transfer phase: This is when data is actually transferred between the end points. Because the switches have the relevant state in them, the forwarding step is easy: they know that any piece of data coming from a particular incoming interface is destined for a particular outgoing interface, and just forward it on the appropriate link (notably, they *don't* know the data is from A and destined for B; they just have a mapping between incoming and outgoing interfaces).

   Note: Last week, I told you that these units of data were called
   packets. In the context of circuit switching, they are called
   frames. You will see why later today.

3. Teardown phase: Once the data transfer is done, the state in the
   switches is deleted.

Now suppose two conversations need to share the same link. In circuit switching, we'll use a technique that is remarkably similar to TDMA: TDM (Time Division Multiplexing). In fact, it's so similar, that you might not immediately see any difference.

In TDM, a switch that is splitting its outgoing link between N connections will divide time into N slots. In slot 0, it will send a frame for conversation 0. In slot 1, a frame for conversation 1, etc. This mapping is done in the setup phase of circuit switching, and forwarding at a switch just involves a table look-up.

The differences between TDM and TDMA are subtle. TDMA is a protocol for channel access: nodes only have access to the channel in their respective time slot. TDM is a protocol for multiplexing a channel: nodes might send at any time, but the switch will only forward their data in their respective slot.

But TDMA had one problem: it was bad when data arrived in bursts. The same problem extends to TDM, and thus circuit switching. So one con of circuit switching:

1. Bad when data arrives in bursts.

Assuming the that the link we're sharing can handle C bits/sec, and that each connection requires R bits/sec, we have a second (related) con:

2. Bad when communications send more/less than R bits/sec, or when we
   have more than C/R communications.

Additional concerns:

3. Without some additional work, we would not be able to have two
   conversations from point A at the same time.

4. If a link in the middle of the network goes down, all circuits
   using that link are broken; there is no way to recover given what
   we've seen so far.

So far, it doesn't seem like circuit switching is a great idea, and indeed, packet switching was designed to handle some of these problems. Once we get through our discussion of packet switching, however, you'll begin to see some pros of the circuit switching design.

==========
Approach 2: Packet Switching
==========

Packets

Although it may seem that the first issue with circuit switching is the biggest -- that it only works under very uniform load -- it was

actually the fourth that inspired the design of packet switching.

History lesson: In the late 1950s, Paul Baran envisioned a communication network that could survive a major enemy attack. He came up with three different topologies: centralized, decentralized, and distributed.

The distributed version was deemed to be the most resilient to attacks. If a single node went down, there would be lots of alternate paths for other nodes (in fact, quite a few nodes could disappear and this would still be true). But how would we do that? How would the network "find" these alternate paths, and how would it get the data there? (Side note: this communication network would later become the Internet)

> Historical aside: the fact that the Internet was designed for resilience to failure has myriad implications today. Were we re-designing the Internet from scratch, we might put, e.g., security concerns at the forefront. Take 6.033 for more information.

We will talk about how we deal with finding paths in a later lecture, so let's concentrate on the second problem. As it stands, our frames just contain some data; there is enough state in the network to know how to get the data through the circuit set up between the source and destination.

Most importantly, for our discussion, there's nothing in that frame that tells the network where it is headed. This is okay if we're in a circuit switching world where the circuit always exists. But if we're trying to be more resilient to failure, we need something more. We need packets.

A packet contains a header and data. The header is going to contain, at a minimum, the source *and* destination address:

```
[  Source address   ] [ Destination address   ] [  Data  ]
^                                               ^
|_________________ header ____________________|
```

> Aside: Now you can see why I referred to the data units in circuit switching as frames. In networking, the term "packet" implies that a destination address is attached.

Now, our switching mechanism is different. There is no circuit to setup and teardown. Instead, we do per-packet forwarding: at each node, the entire packet is received, stored, and then forwarded (store-and-forward networks). Note, then, that there is no explicit allocation of resources (the switches don't make a decision, e.g., to give each connection 1/Nth of the outgoing link capacity).

In getting all this to work, our packet header needs to contain a few more things. Right now, we'll just add a length field, to give the length of data in the packet (remember: packets can be different sizes, although they're often about 1.5KBytes):

```
[ Src addr ] [ Dst addr ] [ Length ] [ Data ]
^                                    ^
|___________ header _________________|
```

> Aside: For reference, packet headers in the Internet are much more

complicated. The IPv6 header has 8 elements and is 300 bits long. The TCP header has at least 17 elements and is at least 160 bits long. We won't study these headers in 6.02.

So what will happen if a node or link in the network goes down? Assuming the switches can find an alternate route (they can -- we'll discuss this in future lectures), there is enough information in the packet to get it to its destination. It might take a little bit longer than normal, but there's still a high probability that it will make it.

Switching

How does the actual switching work? It's very simple: a bunch of packets will arrive at a switch, and they will be stored in a queue before being forwarded. If the queue is full, an arriving packet will be dropped. We'll talk more about queues later today; for now, assume they're very large (they aren't; just assume that they are). Finally, if there are multiple outgoing links, we determine the correct outgoing link with the information in the packet destination (this step is known as demultiplexing).

So what's so great about all this, besides our newfound ability to route around failures? There are a few other good things:

1. Packet switching doesn't was the capacity of links, because links can send any packet that is available (i.e., we don't waste time slots on connections that aren't currently sending. Of course, if *no* connection is using a particular link, then that link won't have a packet to send.)

2. There's no setup/teardown phase, which means there's no overhead in that regard. This is important for small connections, when the overhead of setting up/tearing down a circuit can be overwhelming compared to the duration of the connection.

3. It can provide variable data rates on an "as needed" basis.

But there is one last thing that is really cool: "statistical multiplexing". Aggregating multiple sources of traffic smooths out bursts in the traffic. This is similar to what we saw in part 2 of 6.02, where taking the mean of M independent random samples decreases the ratio of standard deviation to the mean.

(Students: See slides for this)

Why do we care about smoothing bursts? Imagine that you are a network engineer, and you are building a new network. You have some measurements that indicate the following:
- Over 10 millisecond windows, the highest burst of traffic is 50,000 bytes (i.e., 50,000 bytes / 10 milliseconds is the highest throughput)
- Over 100 millisecond windows, the highest burst of traffic is 200,000 bytes (i.e., 200,000 bytes / 100 milliseconds)

So you have two design choices:
1. Provision the network such that it can handle 50,000 bytes every 10 milliseconds.
2. Provision the network such that it can handle 20,000 bytes every 10 milliseconds.

Option 1 allows you to handle all bursts of traffic. However, a lot of the time this network will be overprovisioned: there will be more capacity in the network than there is traffic.

Option 2 allows you to handle *most* bursts of traffic, but not all (note 20,000B/10ms = 200,000B/100ms). Option 2, however, is much cheaper than Option 1. What could we do to the network to handle the occasional bursts in traffic? Add queues! Queues absorb bursts.

So with smoothed-out bursts of traffic, we can provision our networks to better match the average demand, and then add queues to absorb any bursts.

==========
Queues
==========

Sources of Delay

Now to queues. Queues manage packets between arrival and departure. They are a necessary evil: they're needed to absorb bursts, but they add delay by making packets wait until the link is available. So ideally, they shouldn't be too big.

Determining the right size for a queue can be a challenge. To put this in context, let's enumerate all of the sources of delay in a network.

Imagine this scenario:

Sender ---------- Switch ---------- Receiver

(the switch might have multiple outgoing links)

The sender has a packet of 100 bits ready to send, and the link from sender to switch is able to send 10 bits per second. So here is our first type of delay:

1. Transmission Delay: Time spent sending a packet of size S bits over a link (or links). Our transmission delay for the first link is 10 seconds.

Now, that link will also have a latency associated with it, not just a maximum data rate. The latency relates to the second type of delay:

2. Propagation Delay: Speed-of-signal (light) delay: the time to send one bit over the link.

The difference between propagation delay and transmission delay is subtle. The transmission delay relates to how much data the channel can handle at once (or, sometimes, how much data our network interface can output at once). The propagation delay relates to how "long" the channel is.

So now our packet has arrived at the switch, and is placed in a queue:

3. Queueing Delay: The time a packet spends waiting in a queue. This delay is variable, and based on the size of the queue.

Once the packet is through the queue, the switch will need to figure out which outgoing link to send it on:

4. Processing Delay: Time spent by the hosts and switches to process a packet (looking up information based on the header, e.g.). For 6.02, assume that this value is the same for every packet a particular switch encounters.

   Aside: in real switches, packet processing can happen before a packet is placed in a queue, because frequently there are multiple outgoing queues in a switch. This is not important for 6.02, I just don't want you all to go around saying that queueing delay must be experienced before processing delay.

Just to complete our example, this packet would experience the propagation and transmission delay of the second link before reaching the receiver.

All of these delays are fixed for a particular link except queueing delay. So how can we even hope to estimate the time it would take a particular packet to get from one end point to another?

Little's Law

We can use Little's Law! Little's Law will tell us some things about the average time a packet spends in a particular queue.

First of all, a tricky part about queues: there are, in general, two rates associated with a queue. The rate at which things -- in our case, packets -- arrive in the queue, and the rate at which things are taken out of the queue (or "drained" from the queue). If packets arrive faster than they are drained, the queue will grow without bounds; we say it's unstable. In 6.02, we're going to assume that queues are stable unless we tell you otherwise. In a stable queue, the average arrival rate is equal to the average departure rate.

   Aside: why couldn't the arrival rate be less than the departure rate? Because, on average, packets cannot depart from the queue faster than they arrive. Imagine a queue where on average 10 packets arrive per second, and yet 20 packets are drained per second; where are those extra 10 packets materializing from?

Let's imagine a switch that forwards P packets over T seconds (assume T is large). The average rate of this switch -- alternatively, the average rate at which packets are drained from the queue -- is P/T. Call that lambda.

    lambda = average rate of switch
           = P/T in this example

Suppose I plotted n(t), where n(t) is the number of packets in the queue at time t. It might look like this:

Let's call A the area under n(t). What does A represent? The aggregate number of packets in the queue over time T. So what about A/T? A/T is the mean number of packets in the queue at any given time. Let's call that N.

     N = mean number of packets in queue
       = A/T in this example

Now, since A is equivalently the sum of all the queue lengths over time, A represents the aggregate delay experienced by all packets. So we can write an expression for the mean delay per packet: A/P. Call that D.

D = mean delay per packet
= A/P in this example

And now check out this relationship:

N = A/T = (A/P) * (P/T) = D * lambda = lambda * D

N = lambda * D is Little's Law. It tells us that if we know the average number of packets in a queue, and the rate at which packets are drained from the queue, then we can calculate average delay. Or alternatively, if we know the rate at which packets are drained, and the average delay, we can calculate the average number of packets in a queue.

You should notice, naturally, that if N = lambda * D, and we fix lambda, increasing N will increase D. That's exactly what should happen: the more packets in the queue, on average, the longer the delay, on average.

Best Effort Delivery + Layering

One last thing to say about packet switching, which leads us nicely into the rest of 6.02 Part 3.

Part 2 of 6.02 taught you all about what goes on physically in the channel. The "physical layer", in some sense.

[Physical layer: signal processing]

On top of it, we wrapped data into bits, and then into packets:

[Packets]
[Bits]
[Physical layer: signal processing]

From our discussion of queues, you should see that packet switching gives no guarantees: Packets can get dropped due to congestion in the network (queues are too full). They can also get dropped due to corruption in the network (imagine a bit error in the packet header, such that the header no longer makes sense!). Compare this to circuit switching: no packets were ever dropped due to congestion.

To deal with the problem of dropped packets, we add in a protocol that provides reliability: packets are assigned numbers, and the receiver lets the sender know which packets it has received:

[Reliable Transport]
[Packets]
[Bits]
[Physical layer: signal processing]

This reliability layer adds two more problems:
- Since each packet is individually routed, they may arrive at the destination in a different order from that in which they were sent.

- There is no guarantee on latency: delays can vary from packet to packet. If a packet is taking too long, the sender will retransmit it, which can lead to duplicates in the network.

This lack of guarantees that packet switching provides is known as "best effort" delivery. A best effort network tries its hardest to get data there, but provides no guarantees.

Future 6.02 lectures will expand on reliable transport.

Summary

With this discussion complete, you're fully armed to see the pros/cons of circuit switching and packet switching.

Circuit switching:
- Guaranteed rate
- Link capacity is wasted if data is bursty
- Establishes a path before sending data
- All data in a single connection follows one path
- No reordering; constant delay; no dropped packets due to congestion

Packet switching:
- No guarantees (best effort)
- More efficient
- Send data immediately
- Different packets might follow different paths
- Packets may be reordered, delayed, or dropped

So even though circuit switching seemed like a bad idea earlier, you should have a sense of how complicated it's going to be to fix its problems. Packet switching requires the use of queues, which cause all sorts of issues.

Despite this, packet switching is what the Internet uses, so it's what we're going to study from now on.