6.033 Lecture 10 -- Network Layering and Reliability

Last lecture -- saw some of the challenges of building a network at the scale of the Internet:

Universality
Reliability (best effort packet oriented) <--- today
Sharing + Congestion <--- next time
Scalability (+ autonomy) <--- next week

Today -- network organization, reliability

Layering -- form of abstraction. A vertical hierarchy where service provided by a layer is based solely on the layer below

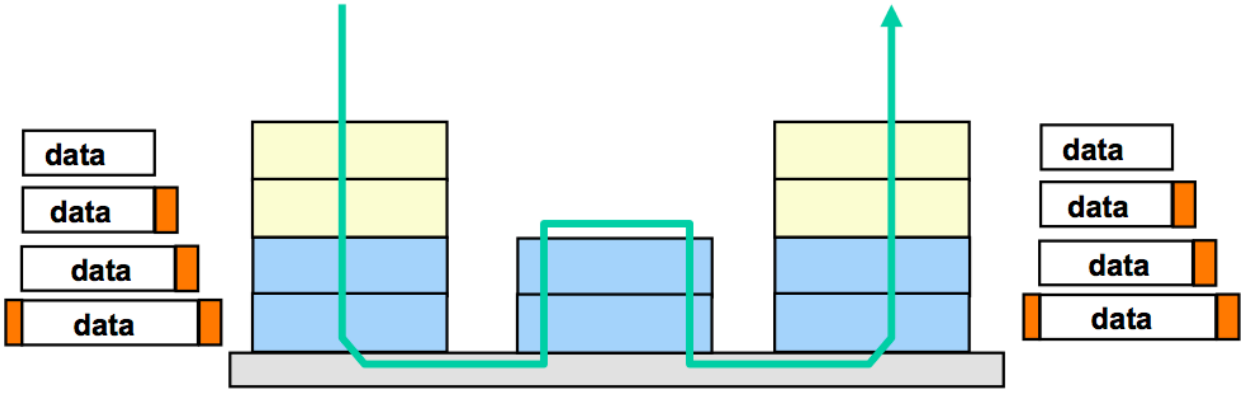
Encapsulatiom:

Each layer accepts/sends a transparent bundle of data from/to the layer above

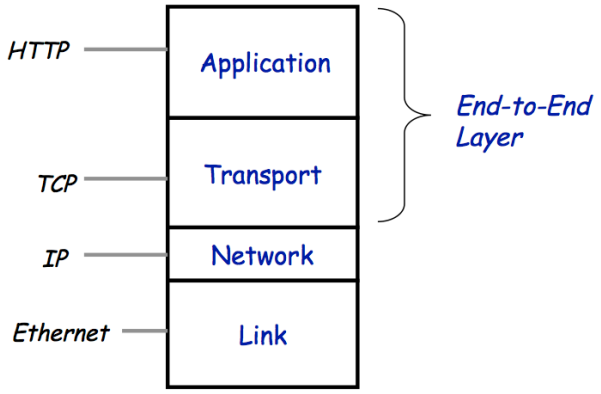
Each layer may split higher level data into a smaller packets

Each layer may service multiple higher level layers

Diagram:



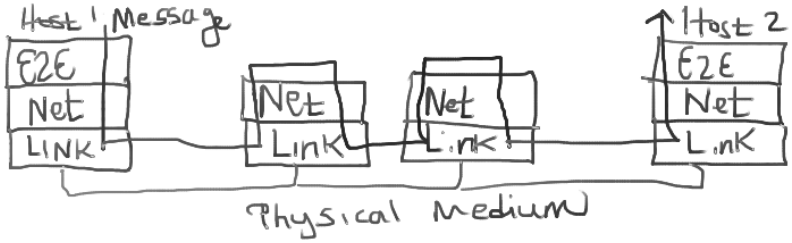
This is the standard design used for the Internet. IP is the "narrow waist" -- all traffic is based on IP, but there are many different Transport and Application protocols, and many different link protocols.



The 4-layer Internet model

Where do these layers run?

E2E/Application layer only run on *hosts* that are communicating; network and link layers run inside of the network too. Diagram:



Saw link layer example in Ethernet last recitation, plus in 6.02 if you took it.

Network layer interface

Recall that network layer is **best effort**, meaning:

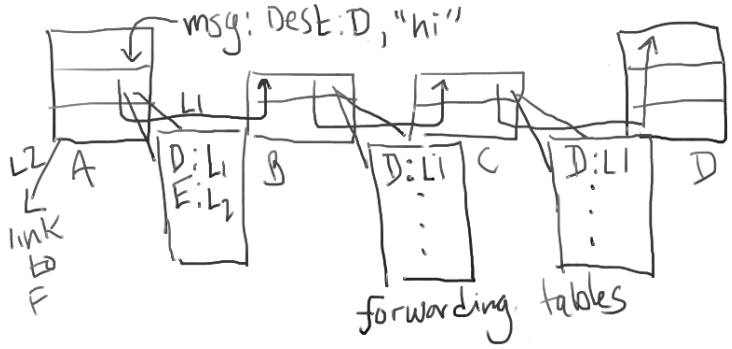
- packets may be lost, reordered, garbled
- losses due to no route, queue overflow, packet corruption / collisions, etc.
- reordering due to packets taking different routes, retransmissions

Network layer is also message oriented.

Two main jobs:

Forwarding -- sending data over links according to a *routing table*
Routing -- process whereby routing tables are built

Routing -- compute the forwarding table -- talk about how the Internet does this next week.



Routing layer consults forwarding table, calls link layer with next hop, link layer adds header, sends to next hop; next hop strips off link header, sends up to net layer, which consults forwarding table, ….

E2E layer addresses limitations of Network Layer. Some things applications might want:

1. Connection, stream of data, rather than having to explicitly divide data into messages

2. All packets to arrive (reliable)

3. No duplicates (at most once)

4. Packets arrive in order

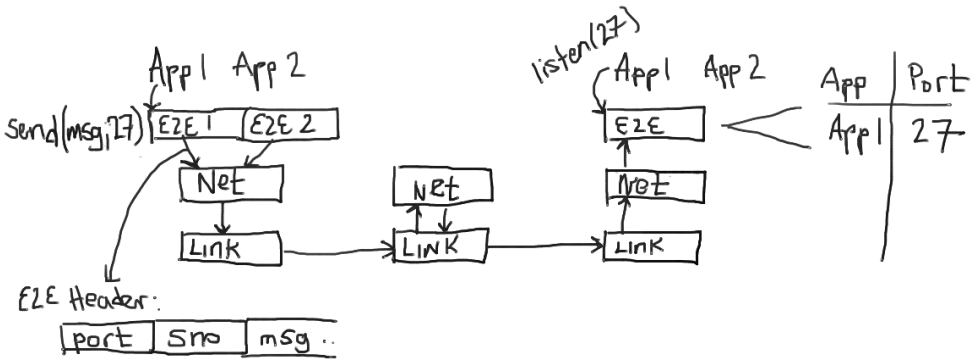
5. Efficiency

Not all applications need all of these -- for example, in voice chat app, you might prefer more predictable latency w/ some packets being dropped, rather than all packets arriving with unpredictable latency.

This list above is roughly what Internet's TCP gives you. Internet provides other protocols, like UDP (none of the above) or RTP (streaming, but not reliable)

Different E2E layers will provide different subsets of these features. Goal is to build these features without modifying network layer.

Diagram: (including port->app table), packets



Streaming connection (add entries to port->app table)

src	dest
conn = open_stream(dest, port)	conn = listen(port)
send(conn, bytes1)	
send(conn, bytes2)	bytes = recv(conn)
close_stream(con)	

E2E layer fragments streams up into packets that it sends to network layer.

Reliability --> At least once

All packets arrive, and no packets are corrupted

For corruption, typically packets have some kind of checksum, which is checked when the packet is received.

How to ensure all packets arrive?

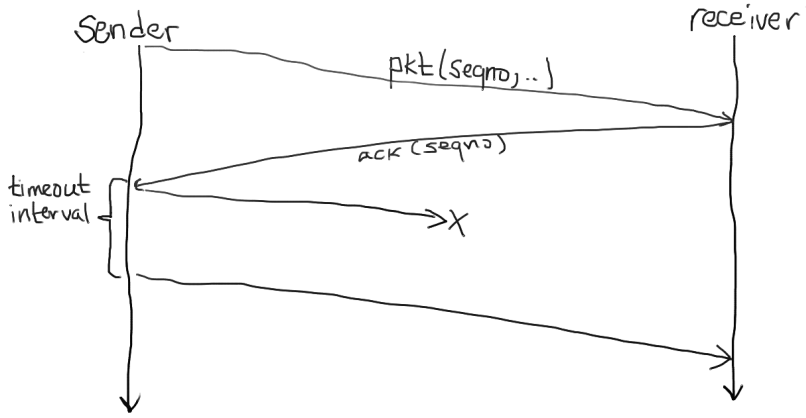
Idea:

have receive send an acknowledgement for every packet. Set a timer on the sender, and when the ack doesn't arrive after that much time, resend.

associate a unique sequence number with each packet.

attach sequence number to the ack.

Diagram:



How to set the timer interval? 1 s? 1ms? Networks vary a lot! In a local wired network, might see e2e latencies < 1ms. In a cellular network, or intercontinenal network, might see latencies > 1s.

Too short --> always resending
Too long --> underutilization of network (show)

So what should we do? Adapt timeout!

Measure round trip time (RTT) and adjust.

RTT = time between packet sent and ack received
Just use one sample? No.
Just use last time + error? No. -- too much variability, even on one network (show slides)

Use some average of recent packets.

To avoid having to keep window of averages around, can use exponentially weighted moving average:

(show slides)

Ok, so now we set the timer appropriately. Ensures that at least one copy of every data item arrives.

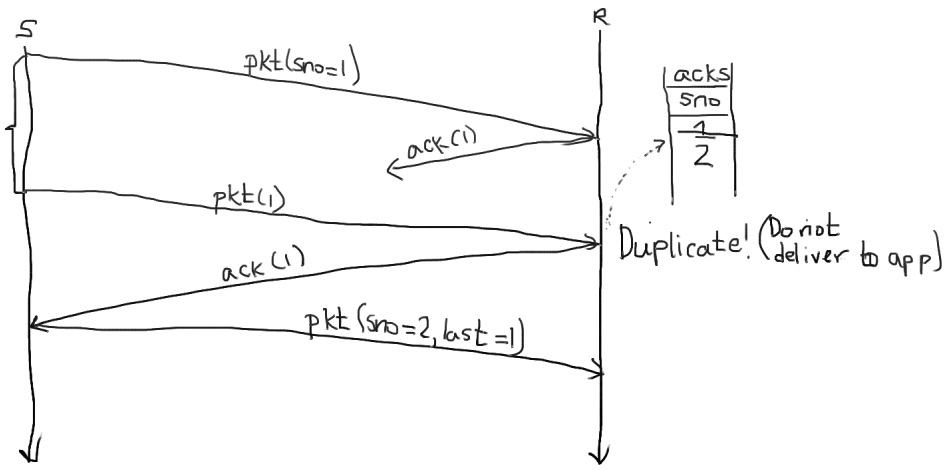
Can we have duplicates? (yes, why?)

What to do about this?

At most once delivery

On receiver, keep track of which packets have been ack'd (in a list). When a duplicate packet arrives, resend ack, but don't deliver to app.

Diagram:



When can you remove something from list of acks? Since acks can be delayed or lost, could receive resent packets quite a bit later. One typical solution is to attach last msg id received on messages from sender.

At least once delivery and at most once delivery together are **exactly once delivery**.

Bit of a misnomer.

Issues:

- delivery may not be possible
- Suppose receiver crashes after receiving message but before sending ack; reboots --> did it process the message or not, what if it receives message again?

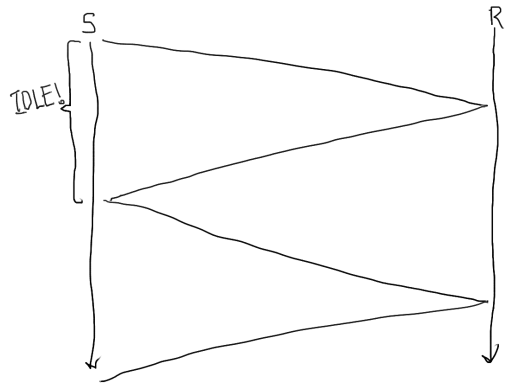
We will talk much more about building reliable systems and these issues after spring break.

- What does an ack mean? Typically just that the E2E layer handed the packet off to the application, *not that the application processed the message.*

Note that the protocol thus far deliver stuff in order b/c it only has one outstanding message at a time.

At this point, we've built a streaming, exactly once, in order delivery mechanism. Problem is it is SLOOW.

Show diagram:



Suppose RTT is 10 ms. Can only send 100 packets / second. If packets are 1000 bits, then we are only sending at 100 Kbit/sec. Not good!

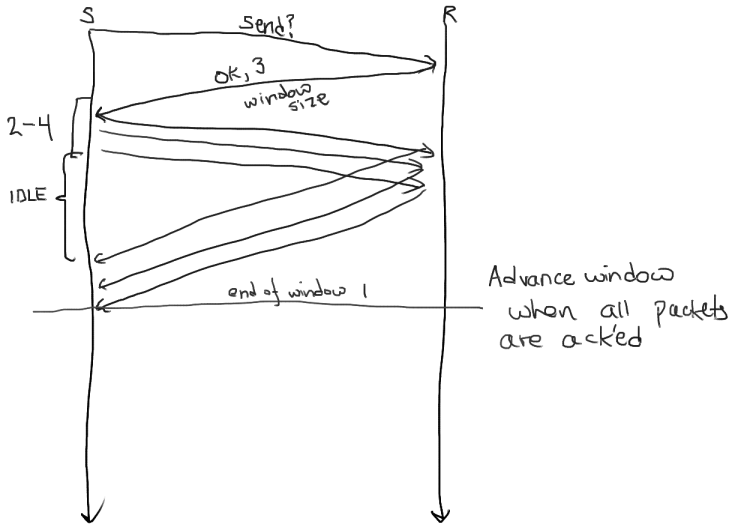
Can we just make packets bigger?

How big would they have to be for gig-e? 10,000 times bigger --> 10 mbits. Huge! Retransmissions are super expensive, and we aren't doing a very good job of multiplexing the network.

Solution: smaller packets, but multiple outstanding at one time.

"window" of outstanding packets

Show diagram:



Still wasteful, since sender waits an RTT. Solution: slide window -- when ack arrives, advance window by one.

(show slides)

Note that if the window size is too small, may still end up waiting.

So, how big to make windows? Want to be able to "cover" delay.
Suppose we have a measure of RTT. (delay)

And suppose we know the maximum rate the network can send (either because of links, delays in the network, or limits at the sender or receiver), (bandwidth)

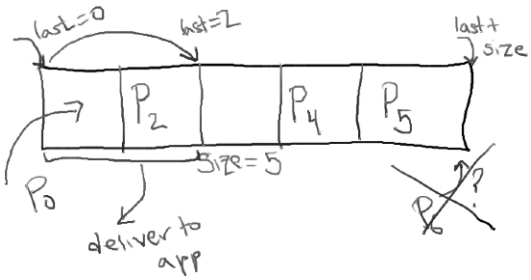
Max packets outstanding is then

window = RTT (delay) x rate (bw)

"bandwidth delay product"

Of course, this assumes we can measure the rate accurately, which turns out to be hard because rate depends on what is happening inside the network (e.g., what other nodes everywhere else inside the network is doing.) This is the topic for next time.

Reordering -- packets may arrive at the receiver out of order




```
recv(p)
        slot = p.sno - last
        if (slot > last + size)
                drop
        else
                new = slot is empty
                if (new) put p in slot
                ack p
        if (slot == last and new)
                deliver prefix to app
                slot += size(prefix)
```

Cumulative acks

So far, have presented acks as being *selective* -- for a specific packet

In Internet, sometimes *cumulative acks* are used

in figure above, when p1 arrives, would send ack(2), indicating that all packets up to 2 have been received.

+ insensitive to lost acks
    (if using selective acks, a dropped ack requires a packet retransmit. Here, if the ack for P2 was lost, when P0 arrives, receiver knows P2 arrived.)

- sender doesn't know what was lost

    here, sender has sent p3, p4, and p5; p4 and p5 have been receiver, but only receives an ack for up to p2.

    can lead to excessive retransmission

In the TCP, both cumulative and selective acks are used:

    common case, use cumulative, so that a lost ack isn't problematic
    when acking a packet for which later packets have arrived,
        use selective acks, so that sender knows what has arrived

Next time -- what if receiver can't keep up with sender?

Congestion control.