Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.892 Fall 1999 **Lecture 1: Single Link Communication** September 8, 1999

Overview. In this lecture, we'll look at the problems that arise when we hook two computers together using some communication medium and try to exchange data between them. We'll start with point-to-point media where the only computers are at the two ends, and then move on to shared media (such as Ethernet). We will conclude with a discussion of bridges. In layering terms, this class covers the essentials of "layer 2," the link layer. It also touches upon what "layer 1," the physical layer, does.

What you should already know. Familiarity with protocol layering will be useful, but not essential. You should know what the end-to-end argument means. Both parts of it!

Readings. MB76; PD96 (Chapter 3) will be useful.

1 Issues

We will take the seemingly trivial problem of making two computers hooked together "talk" to each other, and convince ourselves that it isn't as trivial as it sounds. Then, we will systematically convert this specification into a set of five tasks and go about solving each one. In the end, we will see how these all fit together.

The first issue is that physical media (such as phone lines, cable, or ethernets) transmit analog signals, not bits. We therefore need a way of converting from bits to signals. This is typically called *modulation*, and unsurprisingly, the converse process of analog-to-bits done at the receiver is called *demodulation.* Among the variety of things done at the physical layer, this is an important step.

Recall that what we really want to do is to send a big file across to the other computer. Assuming that we've solved the (de)modulation problem, we know how to send bits across. We need to way of packaging portions of the file and shipping those bits across efficiently. One approach would be to treat the file as a bit-stream and continually ship bits across. Another would be to "packetize" or make *frames* of smaller sizes and ship them across. This is the *framing problem.*

The laws of physics make noise-free communication impossible under all circumstances. In many cases, we're interested in ensuring that what the receiver sees is precisely what the sender sent it. This means that the receivers needs machinery to perform *error detection.*

In some cases, it might be enough for the receiver to detect errors and throw away that chunk of information. [Can you now see why framing is useful? Can you think of two situations where throwing bad chunks away and not bothering to do anything about it might be the right thing to do?] However, there are many others where it isn't enough just to detect errors, it is important to *recover* from them (can you think of such an application?). This is the *error correction* or *error recovery* problem.

Finally, there are many media, such as Ethernet, where one can attach more than two computers to a single physical medium. We now have to deal with *sharing.* This is the *media access* or *channel access* problem—the problem of determining who gets to send at any point in time, and how competition for the channel is arbitrated and resolved.

2 Modulation & Demodulation

Scheme 1. NRZ. In the simplest modulation scheme, bit "1" may be sent as a high voltage signal and bit "0" as a low voltage signal. The confusing term "Non-Return to Zero" (NRZ) is used to describe this scheme. The main problems with NRZ are that consecutive sequences of the same bit (voltage) confuse the receiver—for example, it can't easily distinguish a zero from no signal, and too many consecutive 1s cause the baseline signal to deviate from the true average.

A key requirement of most (de)modulation schemes is that *clock recovery* be easy. Clock recovery refers to the receiver being able to deduce (or "recover") the sender's clock, since the sender sends information (bits) triggered by clock cycles. Intuitively, clock recovery is made easier by frequent 0-1 and 1-0 transitions.

Scheme 2. NRZI. NRZI stands for Non-Return to Zero Inverted. Here the sender stays the same to signal a 0 and changes to signal a 1. Of course, this doesn't solve the consecutive 0s problem, although it does solve the consecutive 1s problem.

Scheme 3. Manchester encoding. In Manchester encoding, the sender transitions from low to high to encode a 0 and from high to low to encode a 1. This ensures transitions on every bit, facilitating clock recovery. While it solves the problems with NRZ mentioned above, it is somewhat inefficient. [Why?]

Scheme 4. 4B/5B This scheme solves the inefficiency of Manchester encoding by introducing extra bits in the data to prevent long sequences of 0s or 1s. You can think of this as adding some redundancy to the data ("coding") so that clock recovery is easy. Specifically, it converts every sequence of 4 bits into a 5-bit code and then proceeds to encode it using NRZI (this is why we even discussed NRZI!). The trick is that the 4-bit to 5-bit encoding ensures that no consecutive sequence of three or more zeroes appear in any valid encoding, thereby allowing NRZI to be used. [How efficient is 4B/5B?]

3 Framing

Examples of framing protocols include PPP (the Point-to-Point Protocol) and HDLC (High-level Data Link Control). The idea is that the sender demarcates the sequence of bits with BEGIN and END markers (well-known 8-bit patterns, 01111110 in the case of HDLC) and the bits in between correspond to a *frame*. The link-layer then ships the frame off to the receiver, using one of the techniques above for modulation. At the receiver, the link-layer has to take these frames and deliver them to the higher layer application that the sender application wanted to talk to. This is the simplest example of protocol layering, illustrating the three interfaces that every layer has: (i) peer interface, to the peer it is communicating with, (ii) lower-layer interface, from which it receives data on the sending side and to which it sends data on the receiving side, and (iii) higher-layer interface, for the converse. The last two interfaces are sometimes called service interfaces.

One problem that arises in such framing is that the BEGIN or END marker might show up in the actual data being transmitted. Left unchanged, this will confuse the receiver into delivering a frame wrongly to the higher layer. The solution to this problem is called *bit stuffing* and is a lot like using a well-known escape sequence. In the context of HDLC and PPP, every time a sequence of 5 1s is observed, the sender introduces a 0. The receiver now uses the following decoding strategy: if it sees 5 1s, then it looks at the next bit. If that's a '0' then it must have been stuffed, so it removes it and proceeds. If that's a '1' then this is either the END, or the frame has suffered an error in transmission. By looking at the *next* bit this is easy to figure out—the next bit *must* be a 0.

This form of framing used by HDLC and PPP is called *bit-oriented* framing. One of the problems with this form is that bit-stuffed frames are variable in length, with the size of the frame depending on the actual payload.

4 Error Detection

There are many ways of detecting errors in transmissions, including parity, checksums and CRCs, in increasing order of sophistication. The challenge is to do this well wth as little overhead as possible. Details are in the 6-page handout from Peterson and Davie's book; I can't improve on their excellent description of this material!

5 Error Recovery

There are two forms of error recovery over noisy or loss-prone channels: ARQ (Automatic Repeat reQuest) and FEC (Forward Error Correction). ARQ uses some form of receiver acknowledgments to perform retransmissions, while FEC schemes use coding theory to add redundancy to the transmitted data, allowing the receiver to correct certain commonly occuring error patterns. We will not study specific FEC schemes in this course (there are other courses at MIT devoted to this topic).

5.1 ARQ

The simplest form of error recovery is via acknowledgments in which the sender sends a packet (or frame), and waits for an acknowledgment (ACK) from the receiver. Upon receiving this ACK, it sends the next packet. If it doesn't receive an ACK within a certain period of time, called the *timeout* period, it assumes that the packet was corrupted or lost and retransmits it. This simple scheme is called *stop-and-wait* for obvious reasons.

One might think that stop-and-wait doesn't need any information in the packet header, for at most one packet can be outstanding at any point in time. However, this isn't true, since the sender can't be absolutely sure that a packet that it presumed lost was *actually* lost (or corrupted)[1]. For example, it could have just been delayed for a time period longer than the timeout period. This means that an ARQ protocol needs a sequence number so that the receiver can distinguish amongst duplicates. A little thought shows that a 1-bit sequence number that alternates ("wraps around") between 0 and 1 will suffice, even when there are several retransmissions for the same packet.

The main drawback of stop-and-wait is that it allows only one outstanding packet in each round-trip time (RTT), implying that the maximum throughput is bounded by the ratio of the size of a packet to the RTT. In other words, it doesn't "fill the pipe" between sender and receiver.

[1] In general, a sender and receiver can't achieve true synchrony over loss-prone networks. This is the famous two-generals problem, where two generals on either side of a valley wish to agree on a common time at which to simultaneously attack an enemy in the valley, by exchanging messages using soldiers on horseback that travel via the same valley. These messages are loss-prone, since unfortunate soldiers could be eliminated or detained for long periods by the enemy in the valley. It is easy to prove this property by contradiction on the length of the shortest protocol that can in fact accomplish this task. Notice that the last message of the shortest correct protocol is redundant since it can in fact be lost, which contradicts our choice of the shortest protocol.

How much data can the sender-receiver "pipe" hold assuming that we know the available capacity (bandwidth, B) and the RTT, d? Suppose this quantity is P. We know that when the pipe if full, the bottleneck capacity, B is fully utilized. We also know that when the number of unacknowledged bytes in the pipe is P and no data is lost, the observed receiver throughput is P/d. Thus, the right pipe size in the absence of any variations of bandwidth or RTT is $B \times d$. This quantity is often called the *bandwidth-delay product* of the network between sender and receiver. Ideally, the sender would like to ensure that $B \times d$ bytes "in the pipe" rather than waiting to see every packet acknowledged before proceeding (unless of course $B \times d < S$, where S is the size of a packet).

This may be accomplished using a *window-based protocol*. Intuitively, one can see that the size of the ideal window is $B \times d$ to achieve the highest link utilization. The slightly non-intuitive part is that this window should *slide*. That is, a protocol where we send a window's worth of packets, wait for an acknowledgment, then send the next window's worth, doesn't work well. [Why not?]. We therefore use *sliding-window protocols* to achieve our goals.

The idea in a sliding-window protocol is that the receiver acknowledges every packet it receives, and the sender *slides* its window by one to the right. At this time, it sends out a new packet to conserve the number of unacknowledged, *outstanding* packets in flight to remain equal to the window size. In most practical protocols, this window is bounded, as a way to perform *flow control* to ensure that the receiver's buffers are never overrun by the sender sending too much data.

6 Shared Media Access

Often, we are able to attach more than two nodes to the same physical medium, as in the popular Ethernet technology. Such media are called shared media and bring up another important problem—*media access*. The problem is figuring out how to resolve contention for the shared channel amongst several contending stations. Protocols that do this are called MAC (for Media Access Control) protocols.

In general, there are several categories of MAC protocols. Centralized protocols rely on a central controller on the network to decide who gains access to the channel at any point in time. On the other hand, distributed protocols do not have any special stations to perform this task, relying instead on decentralized mechanisms. A particularly interesting example is the protocol used in Ethernet, called CSMA/CD (Carrier Sense Multiple Access/Collision Detect) with exponential backoffs. It is a distributed, randomized protocol that has been highly successful.

When a station wants to transmit data, it first senses the carrier to see if the channel is currently active. It does so by sensing the voltage to see if it is higher than the baseband value. If it doesn't detect a carrier, it goes ahead and sends the packet. It then waits a certain period of time (51.2 microsecs for 10Mbps Ethernet) before trying to send another packet, giving a chance to other stations to transmit in the interim.

On the other hand, if it does detect a carrier, it decides to wait until the carrier is free and then transmits the packet. Of course, two or more stations might end up transmitting data at the same time. This leads to a *collision*. Ethernet stations support collision detection (the "CD" part of CSMA/CD); when a collision is detected, the station makes sure to transmit at least 512 bits of the packet and then stops transmitting. [Why does it send at least this many bits?]

Each time a station detects a collision, it assumes that the collided packet has been irrecoverably corrupted. It waits a certain period of time and tries again. The period of time that it waits for

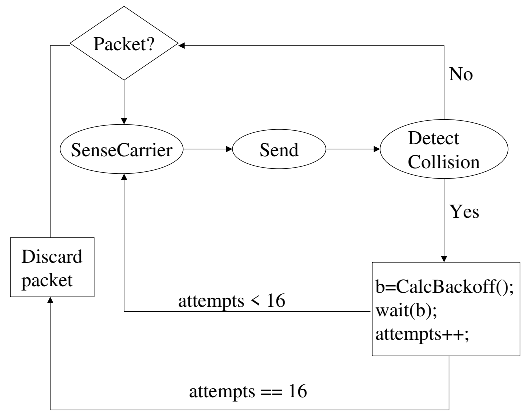

Figure 1: Summary of Ethernet's MAC protocol. The function CalcBackoff() picks a random number uniformly between 0 and the current backoff interval. For every unsuccessful transmission, this backoff interval doubles in size.

is called the *backoff* period. There are many ways of picking the backoff period, and Ethernets use a technique called *exponential backoff*. The idea here is that the backoff value is randomly and uniformly chosen from intervals that successively double in length for every detected collision. The randomization helps avoid station synchronizations, while exponential backoffs improve system stability in the sense that they reduce collision probabilities[2]. The intuition behind using backoffs is that each station tries to estimate how crowded the channel is; the busier it thinks the contention, the longer it backs off. (Because of its distributed nature, no station really knows how many other active stations there are at any point in time.) Stations do not try and retransmit ad infinitum; they give up after some fixed number of tries (usually 16). [How does this follow from the end-to-end argument?] The transmitter algorithm (most of the intelligence is implemented there) is summarized in Figure 1.

The theoretical and practical performance of the Ethernet protocol has been the subject of much research in years past. It isn't hard to show that when packet sizes are small and the number of stations is large (the worst case scenario), the maximum utilization is bounded by $1/e$ (37%). However, in most practical situations, Ethernets function very well, often achieving utilizations as high as 90%. Some simple ways to obtain high utilizations on an Ethernet are to eliminate really long cables (Ethernets can be as long as 1.5 km, but usually there's no need to go that far; shorter segments with bridges work better), use fewer hosts per cable, and use larger packet sizes than the 64-byte minimum.

7 Bridges: Extending the Reach of Shared Media

A single shared medium segment, like a single Ethernet, is limited by the number of stations it can support and by the amount of traffic that can be shared on it. To extend the reach of a single

[2]Actually, if population sizes are infinite, even exponential backoffs don't lead to stability. Fortunately this isn't a problem today.

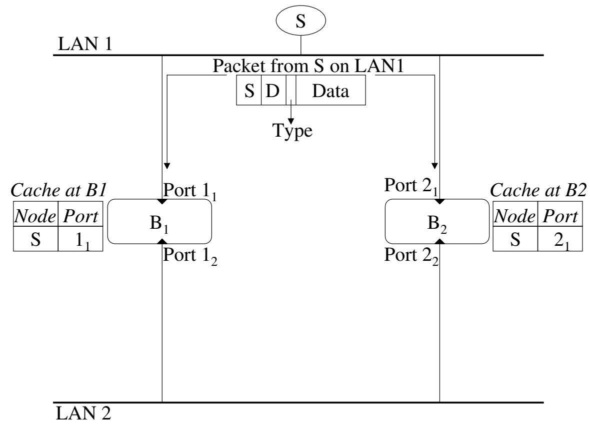

Figure 2: Learning bridges with loops. Such bridges suffer from packet duplication and proliferation.

LAN segment requires some way of interconnecting many of them together. Perhaps the simplest way of extending LANs is via bridges. Bridges extend the reach of a single shared physical medium by sniffing on one segment, copying data, and transmitting them on other segments. Bridges are a great example of "plug-and-play" networking. They are characterized by:

- Promiscuous store-and-forward behavior with two or more ports. Each port has a LAN attached to it, which in turn could contain other bridges. In such a LAN, the aggregate capacity does not exceed the capacity of the weakest segment, since any packet transmitted on any LAN appears on all others, including the slowest one.
- Learning, wherein they learn which stations are on which LAN segments to forward packets more efficiently.
- Spanning tree construction, by which bridge topologies with loops can avoid packet implosions.

Bridges are *transparent* entities—they completely preserve the integrity of the packets they handle without changing them in any way. Of course, they may add a little to the delays experienced by packets and occasionally lose packets because of their store-and-forward nature, but they are transparent in terms of functionality.

7.1 Learning Bridges

The basic idea of a learning bridge is that the bridge learns, by building a cache, which stations are downstream of which port. Then, when a packet destined to a given destination (MAC address) arrives, it knows which port to forward it on. How does it build this cache? It learns by looking at the *source* address of packets it sees. And if it doesn't have an entry for some destination, it simply floods the packet on all ports except the one the packet just arrived on.

This works well, except when there are loops. When there are loops, significant problems arise. Consider the example shown in Figure 2, where bridges B_1 and B_2 have been configured to connect

LAN1 and LAN2 together. [Why might administrators hook up two bridges as shown?] Consider a packet from source S transmitted on LAN1. Since both B_1 and B_2 see this packet, they both pick it up and learn that node S is on LAN1, and add the appropriate information to their bridge caches as shown in the picture. Then, both bridges enqueue the packet to forward it on to LAN2. They compete for the channel (LAN2) according to the CSMA/CD protocol and one of them, say $B1$, wins the contention race. It forwards the packet on to LAN2, and of course this packet is seen by $B1$. $B1$ has now seen a *duplicate* packet! It really has no way of telling that it's a duplicate short of carefully looking through every bit of each enqueued packet, which would be quite inefficient. This is a direct consequence of one of the very reasons learning bridges are so attractive, their transparent behavior. Thus, the duplicate packet would continue to loop around forever (because bridges are transparent, there are no hop limit or time-to-live fields in the header).

But this isn't the worst part—packets in fact *reproduce* over and over! Continuing the above example, $B2$ now enqueues the packet for transmission on to LAN1, while it also has the same packet waiting to be sent out on LAN2. It's not hard to see that this state of affairs will go on for ever and make the system unusable when bridges form loops.

There are several possible solutions to this problem, including avoiding loops by contruction, detecting loops automatically (but not working when loops are present), and making things work in the presence of loops. Clearly the last alternative is the preferred one. The trick is to find a loop-free path in the bridge topology. This is done using a distributed spanning tree algorithm.

7.2 The Solution: Spanning Trees

There are many distributed spanning tree algorithms. Bridges use a rooted spanning tree algorithm that generates the same tree Dijkstra's shortest path trees. The idea is quite simple: bridges elect a root and form shortest paths to the root. The spanning tree induced by the union of these shortest paths is the final tree.

To do this, periodic configuration messages are sent by each bridge to all other bridges on the LAN, consisting of the bridge's unique ID and its distance to the root. By consensus, the bridge picked as root is the one with the smallest ID. Each bridge sends in its configuration message the ID of the bridge that it thinks is the root. These messages are not propagated to the entire extended LAN, but only on each LAN segment; the destination address usually corresponds to a well-known link-layer multicast address corresponding to "ALL-BRIDGES". Initially, each bridge advertises itself as the root, since that's the smallest ID it would have heard about thus far. The root's estimate of the distance to itself is 0.

At any point in time, a bridge hears and builds up information corresponding to the smallest ID it has heard about and the distance to it. The distance usually corresponds to the number of other bridge ports that need to be traversed to reach the root. It stores the port on which this message arrived, the "root port" and advertises the new root on all its other ports with a metric equal to one plus the metric it has stored. Finally, it does not advertise any messages if it hears someone else advertising a better metric on the same LAN.

It is not hard to see that this procedure converges to a rooted spanning tree. Each bridge only forwards packets on ports chosen as part of the spanning tree. To do this, it needs to know its root port and also which of its other ports are being used by other bridges as their preferred (or *designated*) root ports. Obtaining the former is trivial. A bridge deems itself to be the designated bridge for the LAN if it notices that no other bridges have advertised better metrics to the root on that LAN. (When there are ties, the smaller ID is the arbitrarily chosen tie-breaker rule.)

This is the basic algorithm, but doesn't work when bridges fail. Failures are handled by timing information out in the absence of periodic updates. In this sense, bridges treat configuration announcements as *soft state*. The absence of a configuration message from a designated bridge or the root triggers a spanning tree recalculation.

8 Summary

We saw that there are a number of tricky problems to solve even when our network is the smallest imaginable one, running over a single technology: (de)modulation, framing, error detection, and error recovery, and, in the case of shared (but single) media, channel access. We also discussed how to make a single shared medium-based network bigger using bridges, and a cute spanning tree algorithm to avoid loops in this case.

In the next lecture, we'll consider the problem of *internetworking* two or more link technologies into a bigger network. We will also discuss how TCP achieves reliability.