CS 118 **Computer Networks: Data Link Lecture 1, Intro and Framing**

This lecture is a transition from the physical layer to the Data Link layer. We do so in four major parts, that are reflected below using four sections.

- Abstracting the Services of the Physical Layer.
- Functions of the Data Link
- Why Framing.
- Framing Techniques.

1 Abstracting the Services of the Physical Layer

We recalled the 3 sublayer model of the physical layer (coding, transmission, and media). However, when we study the Data Link we can abstract the internal details of the physical layer and consider the physical layer to be a bit pipe.

More precisely, we can model the physical layer as a pipe that connects a sender and one or more (multiple receivers are common in a Local Area Network like an Ethernet) receivers. The sender interface allows the sender to send a sequence of bits; the physical layer "bit pipe" delivers these bits to the receiver interfaces after an arbitrary delay. The delivered bits can be incorrect (i.e., bits can be flipped) or lost. Thus the physical layer is abstracted as a lossy bit pipe that can lose or corrupt bits with some probability; this probability can range from 10^{-4} for telephone lines to 10^{-14} for fibre links. There are two kinds of errors on most physical medium: random bit errors (due to effects like thermal noise) and burst errors that corrupt a group of bits (due to effects like impulse noise).

2 Functions of the Data Link layer

We will understand the Data Link layer in terms of sublayers just as we understood the physical layer. However, the Data Link Layer is best studied separately depending on whether we deal with point-to-point (i,e, 1 sender and 1 receiver) or broadcast (multiple senders and multiple receivers). Many textbooks do a very confused job of separating out these two. We will start with point-to-point links and then cover broadcast links after two or three lectures.

For both point-to-point and broadcast links, the first two sublayers are the same. The first major function of all links (shown as the bottom sublayer) is to convert a stream of bits into units called *frames* using a process called *framing*. Each frame is a group of bits with extra bits that are used to delimit frames and extra bits used for error detection. Next recall that the physical layer

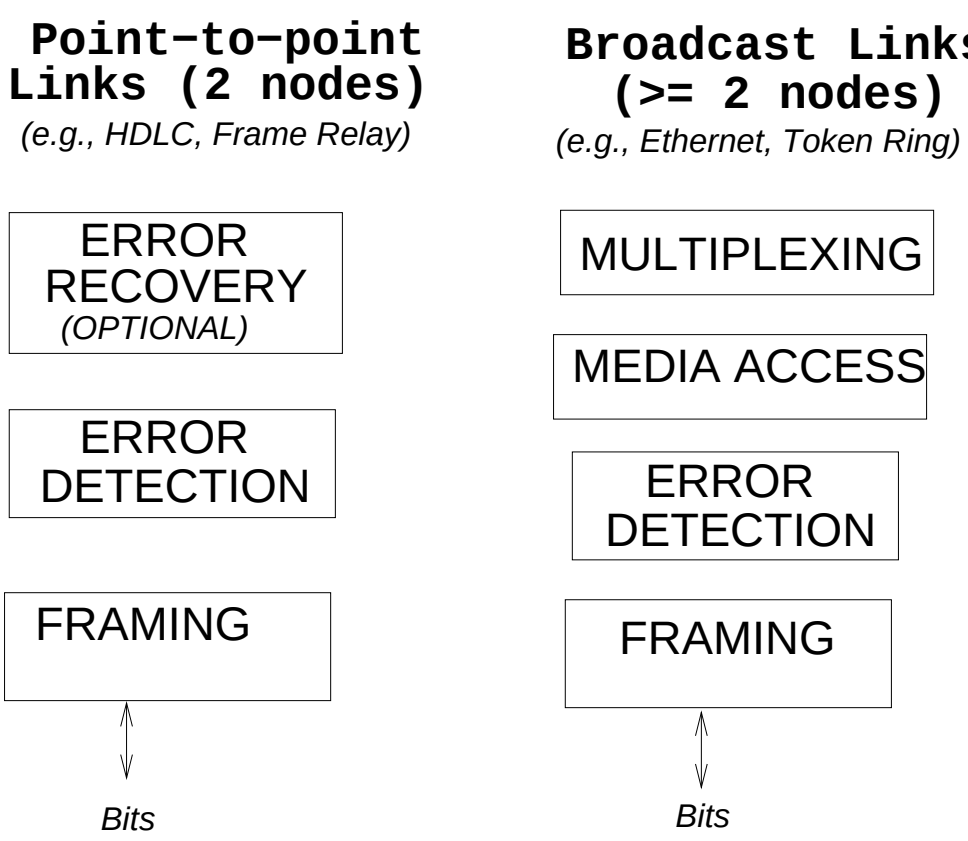

Figure 1: Data Link sublayers for point-to-point and broadcast links

may occasionally corrupt and lose bits. Thus all Data Links offer an *error detection* sublayer that adds check bits to frames in order to detect corrupted frames with high probability. Frames that are detected as being in error are rejected at this sublayer.

Once we get beyond the bottom two sublayers, it is easy to see that broadcast links need something more. Broadcast links have a traffic control problem analogous to a busy intersection where multiple roads meet; without traffic control, different senders could collide. In Data Link terminology, this traffic control is called *media access control* (sometimes abbreviated as the MAC sublayer) because it offers orderly access to the broadcast media. Thus Ethernet and Token Ring protocols need a MAC layer. A point-to-point link with one sender and one receiver does not need media access because there is only 1 sender that can send whenever it wants to. (If we need full-duplex transmission, we assume there is another Data Link in the reverse direction.)

In Figure 1 we see that point-to-point Data Links have an optional layer that does *error recovery*. While error detection merely detects errors in corrupted frames, error recovery takes action upon detection of corrupted or lost frames. This is largely historical and dates back to the days when the physical layer media had very poor error rates (e.g., telephone links). In that case, rather than just dropping frames that were corrupted or lost, it made sense to retransmit lost frames on every network hop. An example is a (still commonly used) protocol called HDLC. However, many new protocols for point-point links (like Frame Relay) dispense with error recovery completely. So do

most broadcast Data Links, because the error rate on such links is quite low.

A typical approach to error recovery is to have the receiver send an acknowledgement (like return receipt for certified mail sent through a post office) to the sender when it gets a frame successfully; finally, if the sender does not receive an acknowledgement within a specified timeout, the sender retransmits its frame. Data links with error recovery are sometimes called *reliable Data Links*; those with just error detection are called *unreliable* Data Links.

Finally, multiple routing protocols may use the Data Link; the broadcast LANs typically allow this using a multiplexing sublayer (sometimes called the LLC sublayer). There is no reason in principle why point-to-point links cannot have such a multiplexing layer, just as there is no reason why broadcast links cannot have an error recovery layer.

Thus Data Link protocols offer five functions: the two required functions are *framing* and *error detection*. *Media access* is required for broadcast links, while *multiplexing* and *error recovery* are optional and can be found in some Data Links but not in others.

For the rest of this lecture we concentrate on framing and error recovery that are common to both layers. But we also discuss why adding error recovery is sometimes a bad idea and sometimes a good idea.

3 Data Link Operation

The main function of framing and error detection is to convert a semi-reliable bit pipe into what we call a quasi-reliable frame pipe. We will define quasi-reliability below. For now, let us examine how the Data Link operates.

The Data Link layer accepts packets from its upper layer (often the network layer); it then adds Data Link header information to a packet to form what we call a frame. These frames are sent to the receiving Data Link bit by bit through the physical layer. The receiving Data Link will assemble the received stream into frames (a process called framing, discussed below). The receiving Data Link will also use information in the frame (e.g., checksums) to decide whether to accept or reject a frame. Accepted frames are passed up by the receiving Data Link to its client layer (typically routing).

All Data Links do *error detection*. The sending Data Link adds extra bits to a frame called a *checksum* (the simplest example is a parity bit). The receiving Data Link examines the received checksum to decide whether the bits in a frame were received in error; if so the frame is dropped. If a receiver does only error detection, there is a finite probability that some frames will be lost.

4 Why Error Recovery is sometimes a good idea

Since error detection by itself doesn't guarantee that frames won't be dropped, shouldn't all Data Links do error correction as well? This question is best answered by the celebrated **End-to-end Argument** that can be stated as follows.

A typical exchange of packets must be sent by the network across multiple hops; thus a packet

will traverse several Data Links and several nodes. The end-to-end argument states that the only worthwhile reliability guarantees are end-to-end guarantees (for example, between the transport protocols at the two endpoints that are communicating.) It may be worthwhile to do hop-by-hop error recovery and detection at each Data Link on the path, but this is only a performance optimization. For correctness, we must rely on end-to-end error recovery.[1]

At first glance, this may seem strange. If every Data Link link on the path does error recovery, isn't every packet that the original source guaranteed to get to the ultimate receiver? So why have end-to-end error recovery? End-to-end error recovery is required for several reasons:

- Each Data Link may successfully deliver the packet to intermediate routers but the intermediate routers may drop the packet.
- The intermediate Data Links may crash in the middle of a packet transfer and the packet can be lost.
- The transport layer must work over both unreliable and reliable Data Links.

In the post office mail analogy, most people prefer to receive a return receipt from the receiver of the letter and not from the next post office in the route! Thus the transport layer must send end-to-end acknowledgements anyway for reliability. Thus any acknowledgements that the Data Link layer are an added bonus that may help performance; however these acknowledgements do not affect end-to-end correct delivery which is ensured by the transport protocol.

How can Data Link layer error recovery help performance? Suppose the path from source to destination goes over 4 Data Links; suppose that each Data Link loses a large fraction of its frames. Without Data Link error recovery, for every frame lost on a single link, a frame must be retransmitted on all four links by the sending transport. It also takes longer to recover from a lost frame because the sending transport must wait for a longer period for an ack from the destination because the delay over 4 hops is larger than the delay over a single hop. With Data Link recovery at every hop, a lost frame causes only a *single* retransmission at every hop and the retransmission is typically quicker.

However, if the physical links have very low bit error rates, then Data Link overhead can actually decrease performance. This is because part of the physical layer bandwidth is wasted due to acks. Also, sending Data Links have to buffer frames for possible retransmission; this adds complexity and slows down the Data Link processing. This tradeoff is illustrated by the following data. On telephone links with poor bit error rates (say $10-4$)[2], it is often a good idea to run a reliable Data Link protocol like HDLC. However, on fibre links or local area networks that have good bit error rates, it pays to use unreliable Data Links (i.e., Data Links that do only error detection) like ATM, Frame Relay, and Ethernet.

To quantify the tradeoff between requiring hop-by-hop error recovery versus removing it, consider the following example. Suppose we have a 3 hop path. First assume we have no errors and we

[1]Note that end-to-end error recovery is very similar to error recovery at a Data Link; only this time the acknowledgements are sent from the ultimate receiver to the original source.

[2]note that if 1 in 10,000 bits are being lost, the probability of a 10,000 bit frame being lost is extremely high

use hop-by-hop error recovery. Then hop-by-hop buys us nothing but we still need acks on every hop. If the acks are 20 percent of the size of a data packet, we still transit 20 percent extra bits. Thus hop-by-hop is not a good idea. However, suppose that the third hop link is losing half its frames. Then each frame needs to be sent twice (on average). In hop by-hop we have to send it only twice on the last hop, but if we have no hop-by-hop we have to send it twice on all three hops. leading to $6n$ transmissions for n data frames. If we have hop-by-hop we only need $4n$ transmissions for n data frames. This results in a savings of 66 percent which can make up for the extra ack overhead. Thus with extremely high error rates it pays to use hop-by-hop while with extremely low error rates it pays to not do hop-by-hop.

5 Quasi-Reliable Frame Pipes

One of the main functions of a Data Link, as the task of converting a somewhat reliable bit pipe, into a quasi-reliable frame pipe. With the previous model of Data Link operation, we can define quasi-reliability as follows:

- The probability of a receiving Data Link passing up an incorrect frame to the client layer should be very, very small. This is known as the *undetected error probability*; a possible goal for a system is one undetected error every 20 years. Typically, this means that we require good checksums that can detect almost all corrupted frames.
- The probability of a receiving Data Link dropping frames sent by the sender should be reasonably small to allow good performance. This is the *frame loss probability*; a typical goal for a fibre optic link might be one frame loss every day.

Why do we stress that the probability of undetected error should be almost zero? Sadly, this is because most transport protocols and user application programs ignore the end-to-end principle when it comes to the integrity of Data. Consider sending a file across the network. The file is broken up into several packets. If one of these packets is corrupted on a Data Link, and the Data Link checksum does not detect this error, then a corrupted version of the file can be received at the destination.

But, you will say, surely the end-to-end argument teaches us that the sending application should also do a checksum of the entire file or the sending transport should checksum each packet. This is correct, but doing checksums in software slows down application performance. Thus end-to-end checksums are often not performed for performance reasons; hence it is good engineering practice to require that the probability of undetected errors on Data Links be low.

6 Why Framing

Why go through the indirection of converting from bit-based transmission at the physical layer to frame based transmission at the Data Link layer. Why can't the Data Link layer just offer a bit pipe to the network layer. Two good reasons for framing are:

- If the Data Link layer offers a bit pipe service, then it is impossible for a Data Link to "time-share" or multiplex its services among multiple clients. For example, many workstations run two routing protocols, OSI and IP. Yet two files sent using these two different routing protocols can be simultaneously in transit on the same Ethernet.

- Frames offer a small, manageable unit for error recovery and error detection. For example, it would be bad to retransmit a large file because of a single bit error; with frames, we need only retransmit the frame.

7 How Framing

Any framing algorithm must allow variable length frames, idle time between frames (the sender must not be required to send continuously), and must not depend on assumptions about client behavior. Also, the framing overhead should be low.

There are three types of framing techniques:

- **Flags and Bit Stuffing:** Use special bit patterns or flags at the start and end of frames. To prevent the possibility of confusion due to user data that contains the flag patterns we use *bit stuffing*. We add bits to the user data whenever to prevent the flags from occuring in data. For example, HDLC uses a flag of 01111110; The HDLC bit stuffing rule is as follows: after any sequence of five 1's in the user data, add a 0. Of course at the receiving end the user data must be destuffed: after any five 1's in the data, the next 0 is removed.

  It is easy to explain the algorithm using sublayering; the stuffing layer first converts user data into data bits that do not contain flags. This is passed to the framing layer that adds flags at either end and gives the resulting frame to the physical layer for transmission bit by bit. At the receiver, the physical layer passes the bits to the deframing layer that assembles frames based on the flags and removes the flags. This will work because the user data has been coded to avoid the appearance of spurious flags. Finally, the frame is passed to the Destuffing layer that "decodes" by removes any stuffed 0's.

  Many students wonder how this works if the data contains a flag. So suppose the data is 01111110. After passing it to the stuffing sublayer the "coded" data bits becomes 011111010. After adding a flags at the start and end, it becomes 01111110011111101001111110. When it gets to the receiver, the receiver deframing layer spots the frame and strips the flags to get 011111010. Then the destuffing sublayer removes stuffed bits to yield 01111110.

  A second example that students wonder about is when the data is of the form 111110. They worry that when such data gets to the receiver, the receiver will incorrectly remove the 0 that follows the five 1's. Of course, the reason that does not happen is that the data bits, after stuffing, becomes 1111100 — i.e., there is an *extra* 0, which is then removed by the receiver.

  We can invent many variants of this stuffing rule. In proving that a stuffing scheme works we want to ensure that the flag does not occur in the encoded data. Consider the sequence of encoded data bits. First you need to argue that a flag cannot occur between stuffed bits. This is usually easy because if it did, there would have been an extra stuffed bit added. The

more tricky part of the argument is to show that the stuffed bit cannot be part of a flag — i.e., you want to argue that a flag cannot occur across a stuffed bit boundary. For example, consider the same HLDC flag but a stuffing rule that stuffs a 0 after getting a 0 followed by five 1's. It appears to be more efficient but it does not work! Suppose we had 0111111111110. The resulting output after stuffing would be 01111101111110. We have created a flag at the end! The problem is that a stuffed bit (which we added because of the first five bits) together with the remaining data bits is creating a flag.

Besides proving that a flag does not occur in the data bits we also need to show that a spurious flag cannot be formed by the last few data bits and the first few bits of the real final flag. For example, consider the flag 01010101 and the stuffing rule that says that you should stuff a 1 after receiving 010101. This rule ensures that you will never create a flag that lies totally among the data bits. However, the data can easily alias with the real final flag to cause a spurious flag. For instance, suppose we have the data 001. Then after stuffing the data stays the same. Finally, after adding flags the bits sent are: 01010101 001 01010101. The receiver will remove the first flag and then receive data bit 0 and then find a flag. Thus the receiver will receive the wrong frame. Thus to finish the proof we need to show that if any suffix of a flag is also a prefix of the flag, then we cannot add data bits to the suffix to form a false flag.

- **Start Flags and Character Count:** Use flags to indicate the start of frame and then a length field at the start of frame to indicate how many bits to the end of the frame. This was used by DEC's old Data Link protocol called DDCMP. Note that bit stuffing is not required.

- **Start and End Flags supplied by Physical Layer:** So far we have assumed that the physical layer only supplies the Data Link layer with the ability to send bits, 0's or 1's. Suppose the physical layer also has the ability to send a third symbol, say "F". Then at the start and end of a frame the sender could add say FFFFFFFFF without any possibility of this symbol occuring in the user data. Note that this extra capability is often easy to provide for physical layers that use a coding sublayer. An example, is 4-5 encoding at the physical layer; every 4 physical layer data bits are encoded using 5 bits to ensure transitions. But this means that there are some 5 bit patterns that never occur in user data that can be used to build framing flags.

8 Principles

In the course of the lecture, we mentioned several principles that will recur throughout the course and are useful ways to think about networks. Each discipline (e.g., physics, chemistry) has its habits of thought; one way to encapsulate these habits of thought is in the form of principles. We will do so through out the course, and finally collect them together before the end of the course.

- **Each layer or sublayer exacts its penalty**: Layering is a wonderful thing because it allows us to get a complex task done; however, there is often a cost to layering. To determine the efficiency at the user level we need to add the costs at each layer. For example, the coding sublayer in the physical layer adds overhead by encoding Data Link bits to ensure transitions.

The framing sublayer in the Data Link layer adds flags for framing; error detection at the Data Link requires checksums.

- **The end-to-end argument:** This argument states that the only worthwhile reliability guarantees are end-to-end guarantees (for example, between the transport protocols at the two endpoints that are communicating.) It may be worthwhile to add hop-by-hop error recovery and detection, but this is only a performance optimization. Similar end-to-end arguments can be made for security (i.e., end-to-end encryption is better).

- **Lower Layers should not depend for correctness on assumptions about how the Higher Layers work:** This is elementary software engineering and holds for arbitrary programs. Thus framing algorithms should never assume that user data will never contain some sequence of bits (which can then be used as a flag to delimit frames).

- **All layers have some common problems to solve**: Frame recovery at the Data Link layer has some remarkable analogies to clock recovery at the physical layer. In general, most layers have to address problems of synchronization (e.g., bit recovery, frame recovery), multiplexing (how can multiple clients share the service, e.g., Time Division Multiplexing at the Physical Layer, Protocol Fields at the Data Link layer), and addressing.

- **The best principles will/should be violated for pragmatic reasons:**

  When transfering a file, the end-to-end argument implies that we should add end-to-end checksums to guarantee the end-to-end integrity of the data and not rely on the hop-by-hop data integrity provided by Data Links. In fact, the most common transport protocol on the Internet is the Transmission Control Protocol (TCP) that allows a transport level checksum. However, since TCP checksums increase processing time significantly, this option is often not used. This increases the reponsibility of Data Links to provide good per hop integrity (i.e., probability of undetected errors should be extremely low). However, the result is a violation of the end-to-end argument.

- **Layering and Sublayering are a Good Way to Solve Protocol Problems:** We saw the use of sublayering to understand how bit stuffing works. Layering is thus not just a way of understanding the structure of existing computer networks; it is also a tool for protocol design. It is really a generalization of divide and conquer used in algorithm design.

  While layered (or sublayered) solutions are often the simplest to understand, they are not the most efficient. For example, it is typically more efficient to merge the flag and stuffing layers into one loop that concurrently looks for flags and removes stuffed bits.

- **Good systems thinking can sometimes lead to simple solutions to hard problems:** Most computer scientists are trained to solve problems in isolation — e.g., sort some numbers, devise a bit stuffing scheme to avoid spurious flags. However, most real systems consist of a number of interacting pieces and problems that are hard (for one piece in isolation) can be solved easily (if a number of pieces cooperate). For example, the problem of avoiding spurious flags can be avoided, without the complexity of bit stuffing, by having the physical layer provide another symbol besides 0 and 1.

This idea can also be looked on as doing a function one time instead of several times at each layer. Since bit stuffing is a form of coding and the physical layer already does coding for transitions, why not have the physical layer code do a little extra work and provide extra symbols for delimiting frames.

9 Reading Assignment

None.