MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2011 (Last update: October 17, 2011)
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 4
Why Digital? Communication Abstractions and Digital Signaling

This chapter describes analog and digital communication, and the differences between them. Our focus is on understanding the problems with analog communication and the motivation for the digital abstraction. We then present basic recipes for sending and receiving digital data mapped to analog signals over communication links; these recipes are needed because physical communication links are fundamentally analog in nature at the lowest level. After understanding how bits get mapped to signals and vice versa, we will present our simple *layered communication model*: messages → packets → bits → signals. The rest of this book is devoted to understanding these different layers and how the interact with each other.

■ 4.1 Sources of Data

The purpose of communication technologies is to empower users (be they humans or applications) to send messages to each other. We have already seen in Chapters 2 and 3 how to quantify the information content in messages, and in our discussion, we tacitly decided that our messages would be represented as sequences of binary digits (bits). We now discuss why that approach makes sense.

Some sources of data are *inherently digital* in nature; i.e., their natural and native representation is in the form of bit sequences. For example, data generated by computers, either with input from people or from programs ("computer-generated data") is natively encoded using sequences of bits. In such cases, thinking of our messages as bit sequences is a no-brainer.

There are other sources of data that are in fact inherently *analog* in nature. Prominent examples are video and audio. Video scenes and images captured by a camera lens encode information about the mix of colors (the proportions and intensities) in every part of the scene being captured. Audio captured by a microphone encodes information about the loudness (intensity) and frequency (pitch), varying in time. In general, one may view the data as coming from a *continuous space of values*, and the sensors capturing the raw data may be thought of as being capable of producing *analog data* from this continuous space. In practice, of course, there is a measurement fidelity to every sensor, so the data captured will be quantized, but the abstraction is much closer to analog than digital. Other sources of data include sensors gathering information about the environment or device (e.g., accelerometers on your mobile phone, GPS sensors on mobile devices, or climate sensors to monitor weather conditions); these data sources could be inherently analog or inherently digital depending on what they're measuring.

Regardless of the nature of a source, converting the relevant data to *digital* form is the modern way; one sees numerous advertisements for "digital" devices (e.g., cameras), with the implicit message that somehow "digital" is superior to other methods or devices. The question is, why?

■ 4.2 Why Digital?

There are two main reasons why digital communication (and more generally, building systems using the digital abstraction) is a good idea:

1. The digital abstraction enables the *composition of modules* to build large systems.

2. The digital abstaction allows us to us sophisticated algorithms to process data to improve the quality and performance of the components of a system.

Yet, the digital abstraction is not the natural way to communicate data. Physical communication links turn out to be analog at the lowest level, so we are going to have to convert data between digital and analog, and vice versa, as it traverses different parts of the system between the sender and the receiver.

■ 4.2.1 Why analog is natural in many applications

To understand why the digital abstraction enables modularity and composition, let us first understand how analog representations of data work. Consider first the example of a black-and-white analog television image. Here, it is natural to represent each image as a sequence of values, one per (x, y) coordinate in a picture. The values represent the *luminance*, or "shade of gray": 0 volts represents "black", 1 volt represents "white", and any value x between 0 and 1 represents the fraction of white in the image (i.e., some shade of gray). The representation of the picture itself is as a sequence of these values in some scan order, and the transmission of the picture may be done by generating voltage waveforms to represent these values.

Another example is an analog telephone, which converts sound waves to electrical signals and back. Like the analog TV system, this system does not use bits (0s and 1s) to represent data (the voice conversation) between the communicating parties.

Such analog representations are tempting for communication applications because they map well to physical link capabilities. For example, when transmitting over a wire, we can send signals at different voltage levels, and the receiver can measure the voltage to determine what the sender transmitted. Over an optical communication link, we can send signals at different intensities (and possibly also at different wavelengths), and the receiver can measure the intensity to infer what the sender might have transmitted. Over radio

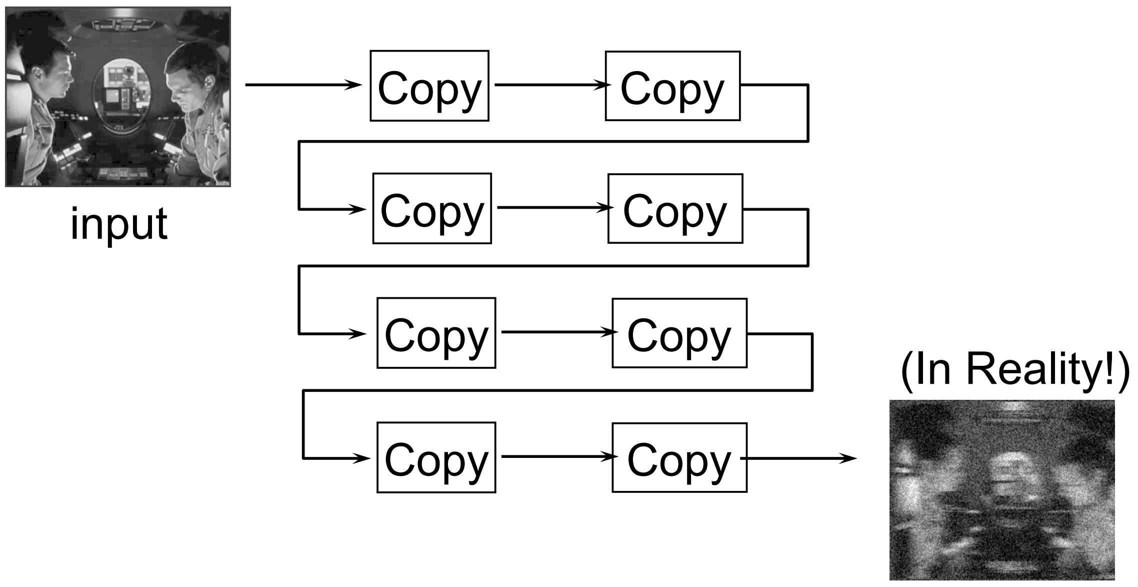

Figure 4-1: Errors accumulate in analog systems.

and acoustic media, the problem is trickier, but we can send different signals at different amplitudes "modulated" over a "carrier waveform" (as we will see in later chapters), and the receiver can measure the quantity of interest to infer what the sender might have sent.

■ 4.2.2 So why not analog?

Analog representations seem to map naturally to the inherent capabilities of communication links, so why not use them? The answer is that there is no error-free communication link. Every link suffers from perturbations, which may arise from noise (Chapter 5) or other sources of distortion. These perturbations affect the received signal; every time there is a transmission, the receiver will not get the transmitted signal exactly, but will get a perturbed version of it.

These perturbations have a cascading effect. For instance, if we have a series of COPY blocks that simply copy an incoming signal and re-send the copy, one will not get a perfect version of the signal, but a heavily perturbed version. Figure 4-1 illustrates this problem for a black-and-white analog picture sent over several COPY blocks. The problem is that when an analog input value, such as a voltage of 0.12345678 volts is put into the COPY block, the output is not the same, but something that might be 0.12?????? volts, where the "?" refers to incorrect values.

There are many reasons why the actual output differs from the input, including the manufacturing tolerance of internal components, environmental factors (temperature, power supply voltage, etc.), external influences such as interference from other transmissions, and so on. There are many sources, which we can collectively think of as "noise", for now. In later chapters, we will divide these perturbations into random components ("noise") and other perturbations that can be modeled deterministically.

These analog errors accumulate, or cascade. If the output value is $V_{in} \pm \varepsilon$ for an input value of V_{in}, then placing a number N of such units in series will make the output value

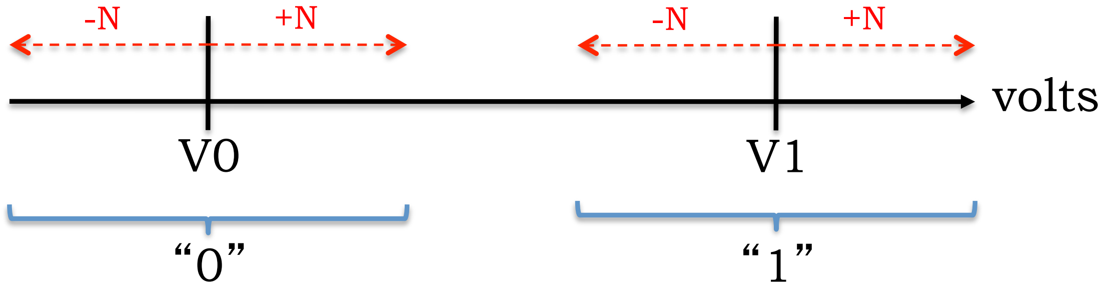

Figure 4-2: If the two voltages are adequately spaced apart, we can tolerate a certain amount of noise.

be $V_{in} \pm N\varepsilon$. If $\varepsilon = 0.01$ and $N = 100$, the output may be off by 100%!

As system engineers, we want modularity, which means we want to guarantee output values without having to worry about the details of the innards of various components. Hence, we need to figure out a way to eliminate, or at least reduce, errors at each processing stage.

The digital signaling abstraction provides us a way to achieve this goal.

■ 4.3 Digital Signaling: Mapping Bits to Signals

To ensure that we can distinguish signal from noise, we will **map bits to signals** using a fixed set of discrete values. The simplest way to do that is to use a *binary mapping* (or binary signaling) scheme. Here, we will use two voltages, V_0 volts to represent the bit "0" and V_1 volts to represent the bit "1".

What we want is for received voltages near V_0 to be interpreted as representing a "0", and for received voltages near V_0 to be interpreted as representing a "1". If we would like our mapping to work reliably up to a certain amount of noise, then we need to space V_0 and V_1 far enough apart so that even noisy signals are interpreted correctly. An example is shown in Figure 4-2.

At the receiver, we can specify the behavior wih a graph that shows how incoming voltages are mapped to bits "0" and "1" respectively (Figure 4-3. This idea is intuitive: we pick the intermediate value, $V_{th} = \frac{V_0+V_1}{2}$ and declare received voltages $\leq V_{th}$ as bit "0" and all other received voltage values as bit "1". In Chapter 5, we will see when this rule is optimal and when it isn't, and how it can be improved when it isn't the optimal rule. (We'll also see what we mean by "optimal" by relating optimality to the probability of reporting the value of the bit wrongly.)

We note that it would actually be rather difficult to build a receiver that *precisely* met this specification because measuring voltages extremely accurately near V_{th} will be extremely expensive. Fortunately, we don't need to worry too much about such values if the values V_0 and V_1 are spaced far enough apart given the noise in the system. (See the bottom picture in Figure 4-3.)

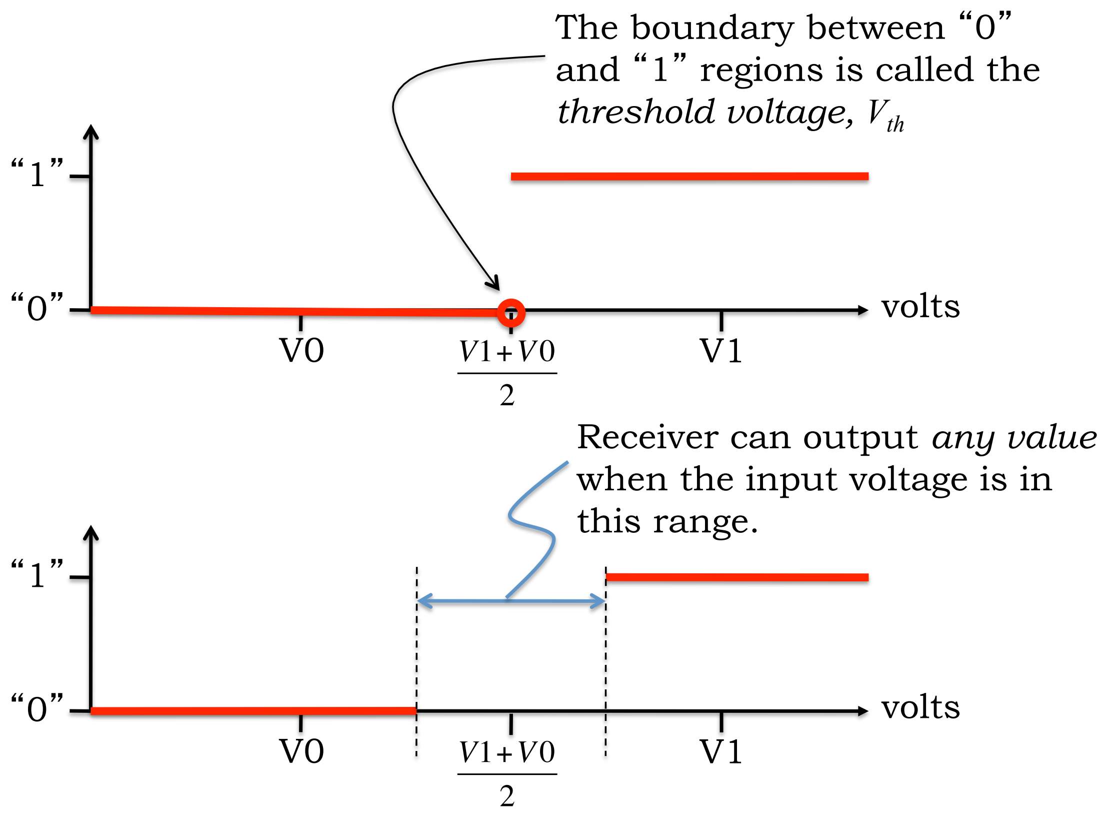
The boundary between "0" and "1" regions is called the *threshold voltage*, V_{th}
"1"
"0"
volts
V0
$\frac{V1+V0}{2}$
V1
Receiver can output *any value* when the input voltage is in this range.
"1"
"0"
volts
V0
$\frac{V1+V0}{2}$
V1

Figure 4-3: Picking a simple threshold voltage.

■ 4.3.1 Signals in this Course

Each individual transmission signal is conceptually a *fixed-voltage waveform* held for some period of time. So, to send bit "0", we will transmit a signal of fixed-voltage V_0 volts for a certain period of time; likewise, V_1 volts to send bit "1". We will represent these continuous-time signals using sequences of discrete-time *samples*. The *sample rate* is defined as the number of samples per second used in the system; the sender and receiver at either end of a communication link will agree on this sample rate in advance. (Each link could of course have a different sample rate.) The reciprocal of the sample rate is the *sample interval*, which is the time between successive samples. For example, 4 million samples per second implies a sample interval o 0.25 microseconds.

An example of the relation between continuous-time fixed-voltage waveforms (and how they relate to individual bits) and the sampling process is shown in Figure 4-4.

■ 4.3.2 Clocking Transmissions

Over a communication link, the sender and receiver need to agree on a *clock rate*. The idea is that periodic events are timed by a clock signal, as shown in Figure 4-5 (top picture). Each new bit is sent when a clock transition occurs, and each bit has many samples, sent at a regular rate. We will use the term *samples_per_bit* to refer to the number of discrete voltage samples sent for any given bit. All the samples for any given bit will of course be

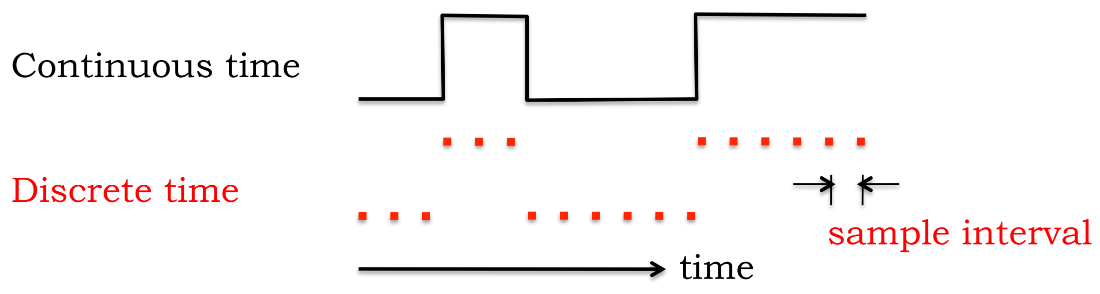

Figure 4-4: Sampling continuous-time voltage waveforms for transmission.

sent at the same voltage value.

How does the receiver recover the data that was sent? If we sent only the samples and not the clock, how can the receiver figure out what was sent?

The idea is for the receiver to infer the presence of a clock edge every time there is a transition in the received samples (Figure 4-5, bottom picture). Them using the shared knowledge of the sample rate (or sample interval), the receiver can extrapolate the remaining edges and infer the first and last sample for each bit. It can then choose the middle sample to determine the message bit, or more robustly average them all to estimate the bit.

There are two problems that need to be solved for this approach to work:

1. How to cope with differences in the sender and receiver clock frequencies?
2. How to ensure frequent transitions between 0s and 1s?

The first problem is one of clock and data recovery. The second is solved using *line coding*, of which 8b/10b coding is a common scheme. The idea is to convert groups of bits into different groups of bits that have frequent 0/1 transitions. We describe these two ideas in the next two sections. We also refer the reader to the two lab tasks in Problem Set 2, which describe these two issues and their implementation in considerable detail.

■ 4.4 Clock and Data Recovery

In a perfect world, it would be a trivial task to find the voltage sample in the middle of each bit transmission and use that to determine the transmitted bit, or take the average. Just start the sampling index at samples_per_bit/2, then increase the index by samples_per_bit to move to the next voltage sample, and so on until you run out of voltage samples.

Alas, in the real world things are a bit more complicated. Both the transmitter and receiver use an internal clock oscillator running at the sample rate to determine when to generate or acquire the next voltage sample. And they both use counters to keep track of how many samples there are in each bit. The complication is that the frequencies of the transmitter's and receiver's clock may not be exactly matched. Say the transmitter is sending 5 voltage samples per message bit. If the receiver's clock is a little slower, the transmitter will seem to be transmitting faster, e.g., transmitting at 4.999 samples per bit.

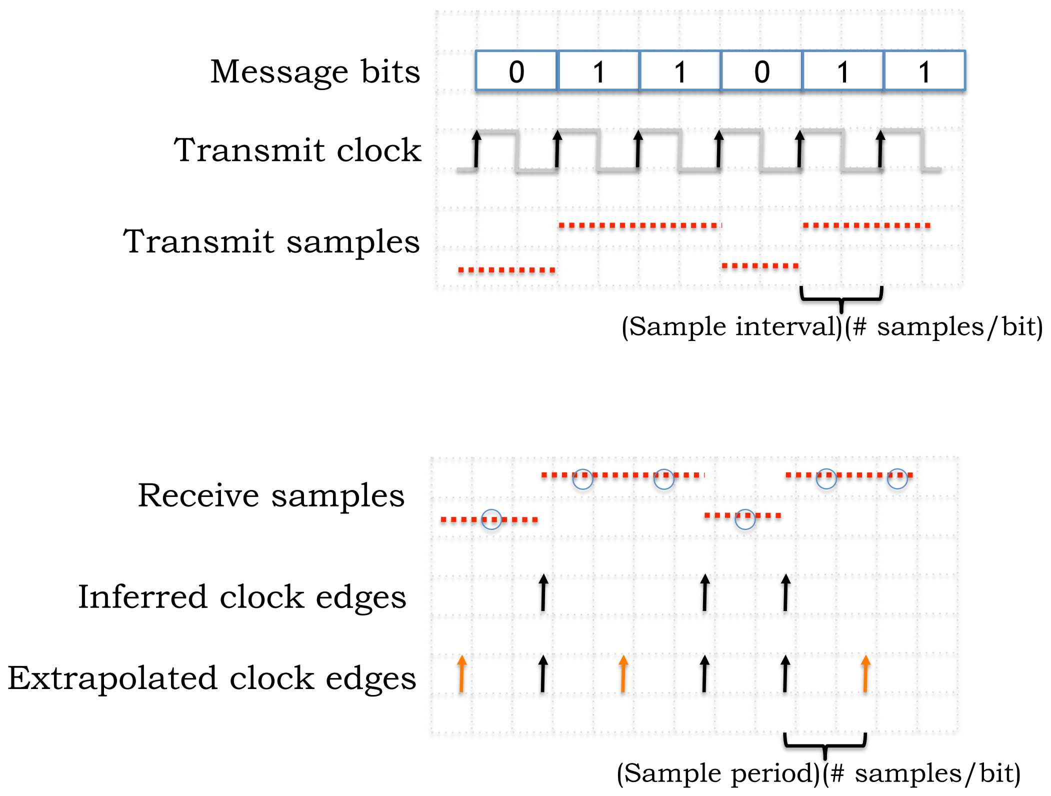

Figure 4-5: Transmission using a clock (top) and inferring clock edges from bit transitions between 0 and 1 and vice versa at the receiver (bottom).

Similarly, if the receiver's clock is a little faster, the transmitter will seem to be transmitting slower, e.g., transmitting at 5.001 samples per bit. This small difference accummulates over time, so if the receiver uses a static sampling strategy like the one outlined in the previous paragraph, it will eventually be sampling right at the transition points between two bits. And to add insult to injury, the difference in the two clock frequencies will change over time.

The fix is to have the receiver adapt the timing of it's sampling based on where it detects transitions in the voltage samples. The transition (when there is one) should happen half-way between the chosen sample points. Or to put it another way, the receiver can look at the voltage sample half-way between the two sample points and if it doesn't find a transition, it should adjust the sample index appropriately.

Figure 4-6 illustrates how the adaptation should work. The examples use a low-to-high transition, but the same strategy can obviously be used for a high-to-low transition. The two cases shown in the figure differ in value of the sample that's half-way between the current sample point and the previous sample point. Note that a transition has occurred when two consecutive sample points represent different bit values.

- Case 1: the half-way sample is the same as the current sample. In this case the half-way sample is in the same bit transmission as the current sample, i.e., we're sampling too late in the bit transmission. So when moving to the next sample, increment the

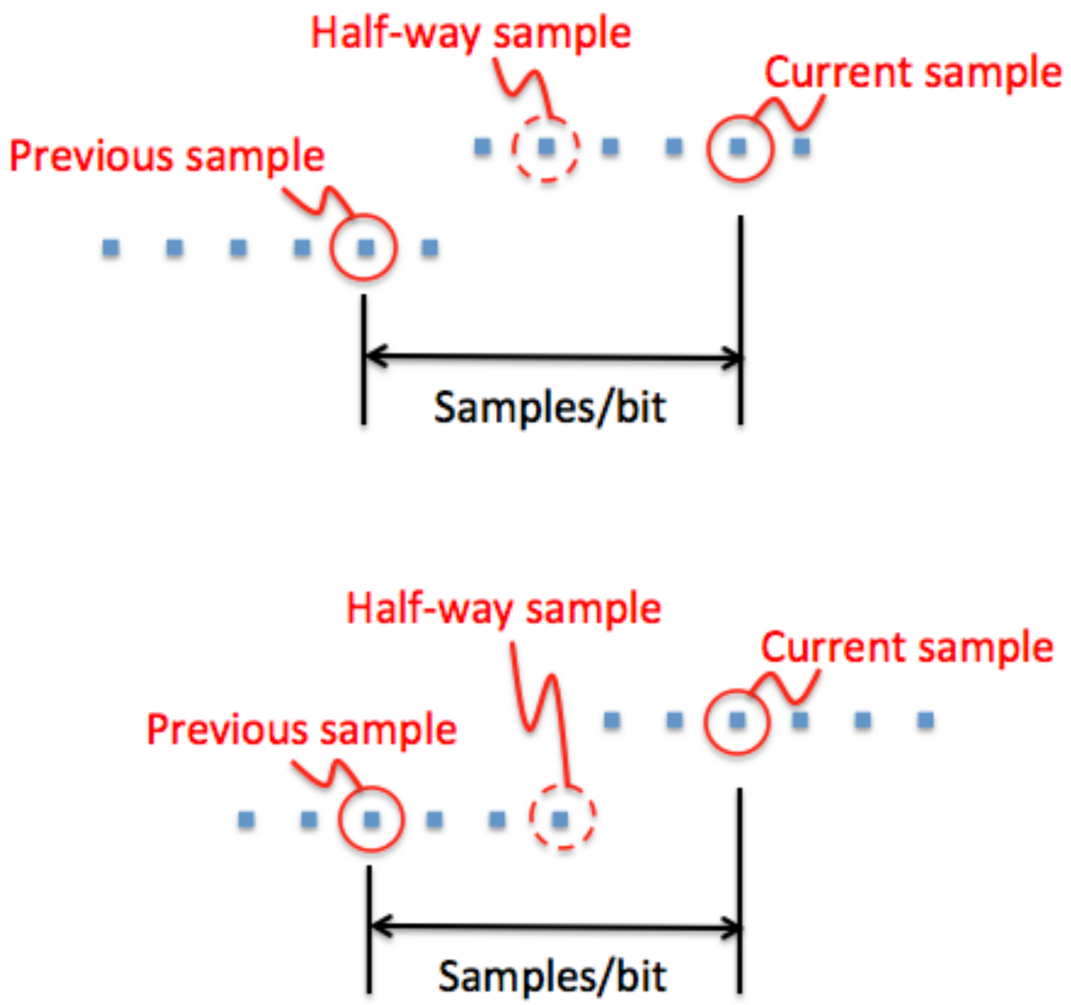

Figure 4-6: The two cases of how the adaptation should work.

index by samples_per_bit - 1 to move ”back”.

- Case 2: the half-way sample is different than the current sample. In this case the half-way sample is in the previous bit transmission from the current sample, i.e., we're sampling too early in the bit transmission. So when moving to the next sample, increment the index by samples_per_bit + 1 to move ”forward”

If there is no transition, simply increment the sample index by samples_per_bit to move to the next sample. This keeps the sampling position approximately right until the next transition provides the information necessary to make the appropriate adjustment.

If you think about it, when there is a transition, one of the two cases above will be true and so we'll be constantly adjusting the relative position of the sampling index. That's fine – if the relative position is close to correct, we'll make the opposite adjustment next time. But if a large correction is necessary, it will take several transitions for the correction to happen. To facilitate this initial correction, in most protocols the transmission of message begins with a training sequence of alternating 0- and 1-bits (remember each bit is actually samples_per_bit voltage samples long). This provides many transitions for the receiver's adaptation circuity to chew on.

■ 4.5 Line Coding with 8b/10b

Line coding, using a scheme like 8b/10b, was developed to help address the following issues:

- For electrical reasons it's desirable to maintain DC balance on the wire, i.e., that on the average the number of 0's is equal to the number of 1's.
- Transitions in the received bits indicate the start of a new bit and hence are useful in synchronizing the sampling process at the receiver—the better the synchronization, the faster the maximum possible symbol rate. So ideally one would like to have frequent transitions. On the other hand each transition consumes power, so it would be nice to minimize the number of transitions consistent with the synchronization constraint and, of course, the need to send actual data! In a signaling protocol where the transitions are determined by the message content may not achieve these goals.

To address these issues we can use an encoder (called the "line coder") at the transmitter to recode the message bits into a sequence that has the properties we want, and use a decoder at the receiver to recover the original message bits. Many of today's high-speed data links (e.g., PCI-e and SATA) use an 8b/10b encoding scheme developed at IBM. The 8b/10b encoder converts 8-bit message symbols into 10 transmitted bits. There are 256 possible 8-bit words and 1024 possible 10-bit transmit symbols, so one can choose the mapping from 8-bit to 10-bit so that the the 10-bit transmit symbols have the following properties:

- The maximum run of 0's or 1's is five bits (i.e., there is at least one transition every five bits).
- At any given sample the maximum difference between the number of 1's received and the number of 0's received is six.
- Special 7-bit sequences can be inserted into the transmission that don't appear in any consecutive sequence of encoded message bits, even when considering sequences that span two transmit symbols. The receiver can do a bit-by-bit search for these unique patterns in the incoming stream and then know how the 10-bit sequences are aligned in the incoming stream.

Here's how the encoder works: collections of 8-bit words are broken into groups of words called a packet. Each packet is sent using the following wire protocol:

- A sequence of alternating 0 bits and 1 bits are sent first (recall that each bit is multiple voltage samples). This sequence is useful for making sure the receiver's clock recovery machinery has synchronized with the transmitter's clock. These bits aren't part of the message; they're there just to aid in clock recovery.

- A SYNC pattern—usually either 0011111 or 1100000 where the least-significant bit (LSB) is shown on the left—is transmitted so that the receiver can find the beginning of the packet.[1] Traditionally, the SYNC patterns are transmitted least-significant bit (LSB) first. The reason for the SYNC is that if the transmitter is sending bits continuously and the receiver starts listening at some point in the transmission, there's no easy way to locate the start of multi-bit symbols. By looking for a SYNC, the receiver can detect the start of a packet. Of course, care must be taken to ensure that a SYNC pattern showing up in the middle of the packet's contents don't confuse the receiver (usually that's handled by ensuring that the line coding scheme does not produce a SYNC pattern, but it is possible that bit errors can lead to such confusion at the receiver).

- Each byte (8 bits) in the packet data is line-coded to 10 bits and sent. Each 10-bit transmit symbol is determined by table lookup using the 8-bit word as the index. Note that all 10-bit symbols are transmitted least-significant bit (LSB) first. If the length of the packet (without SYNC) is s bytes, then the resulting size of the line-coded portion is $10s$ bits, to which the SYNC is added.

Multiple packets are sent until the complete message has been transmitted. Note that there's no particular specification of what happens between packets – the next packet may follow immediately, or the transmitter may sit idle for a while, sending, say, training sequence samples.

If the original data in a single packet is s bytes long, and the SYNC is h bits long, then the total number of bits sent is equal to $10s + h$. The "rate" of this line code, i.e., the ratio of the number of useful message bits to the total bits sent, is therefore equal to $\frac{8s}{10s+h}$. (We will properly define the concept of "code rate" in Chapter 6 more.) If the communication link is operating at R bits per second, then the rate at which useful message bits arrive is given by $\frac{8s}{10s+h} \cdot R$ bits per second with 8b/10b line coding.

■ 4.6 Communication Abstractions

Figure 4-7 shown the overall system context, tying together the concepts of the previous chapters with the rest of this book. The rest of this book is about the oval labeled "COMMUNICATION NETWORK". The simplest example of a communication network is a single physical communication link, which we start with.

At either end of the communication link are various modules, as shown in Figure 4-8. One of these is a *Mapper*, which maps bits to signals and arranges for samples to be transmitted. There is a counterpart *Demapper* at the receiving end. As shown in Figure 4-8 is a *Channel coding* module, and a counterpart *Channel decoding* module, which handle errors in transmission caused by noise.

In addition, a message, produced after source coding from the original data source, may have to be broken up into multiple packets, and sent over multiple links before reaching the receiving application or user. Over each link, three abstractions are used: *packets*, *bits*, and *signals* (Figure 4-8 bottom). Hence, it is convenient to think of the problems in data communication as being in one of these three "layers", which are one on top of the other

[1]In general any other SYNC pattern could also be sent.

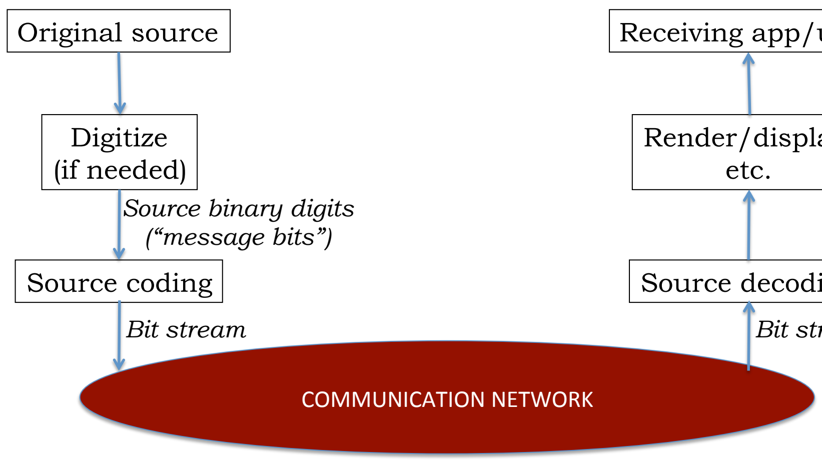

Figure 4-7: The "big picture".

(packets, bits, and signals). The rest of this book is about these three important abstractions and how they work together. We do them in the order bits, signals, and packets, for convenience and ease of exposition and understanding.

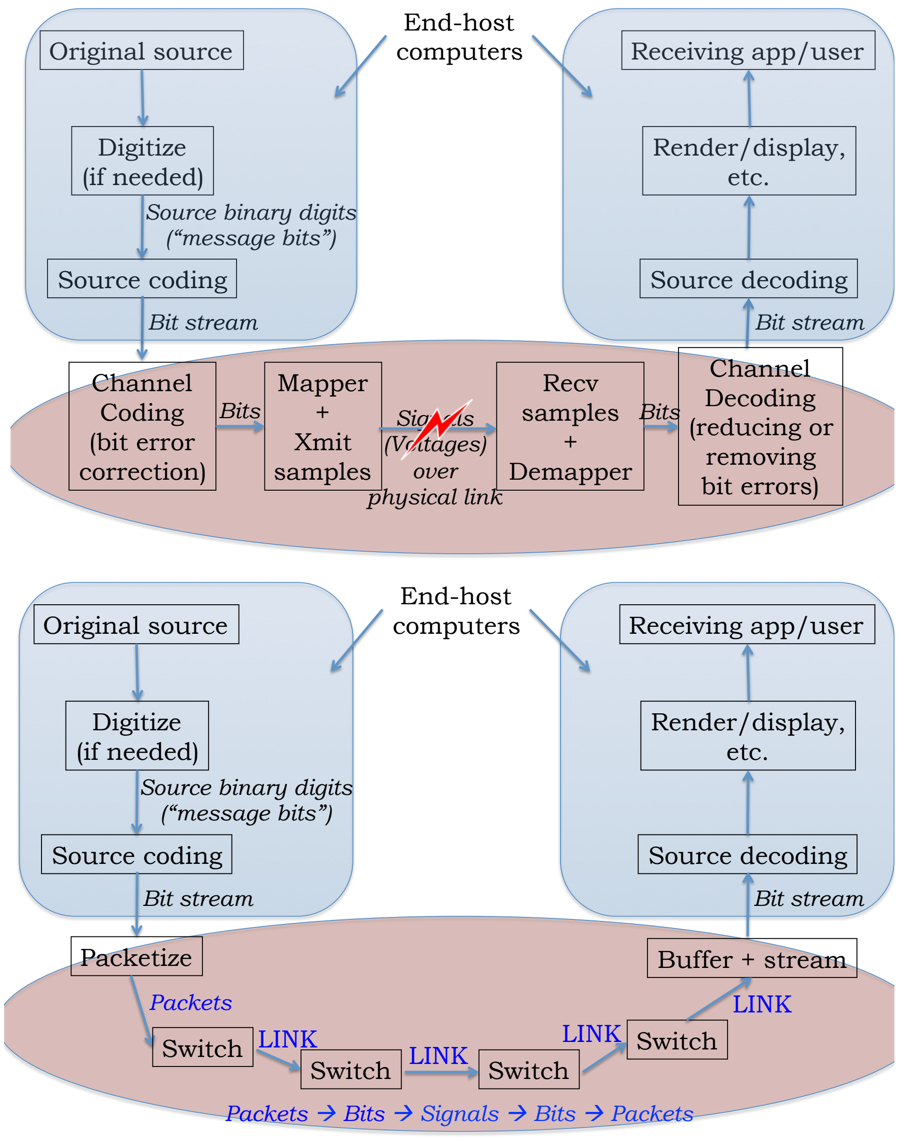

Figure 4-8: Expanding on the "big picture": single link view (top) and the network view (bottom).