6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 14: November 1, 1993

Lecturer: Butler Lampson *Scribe: Rupert Young and Allen Shu*[1]

1 Lecture Overview

This lecture marks the mid-point of the course. The rest of the semester will deal with distributed systems. The lectures of the next couple of weeks will focus on describing what a distributed system is, and what the necessary communications infrastructure is to support such a system. In later lectures, other issues such as fault tolerance in the context of distributed systems will be studied.

The major topics of this lecture are listed below:

- Characteristics of a Distributed System
- ISO Reference Model
- General Principles
- Physical/Data Link
- Networks

2 Characteristics of Distributed Systems

One of the questions to ask in studying distributed systems is what there is about a system that makes it distributed. A naive response to this might be to describe it as a system which has nodes connected by wires, such as in Figure 1. (The terms "nodes" and "wires" by the way are legacies of the "good old days" of computer networking.)

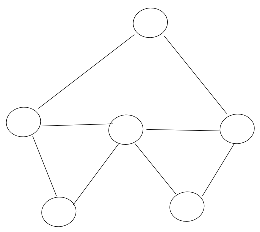

Figure 1: A Distributed System ?

[1] These notes are based on those written by Nikhil Gautam in Fall 1992

However this is not a complete answer. Many systems can be described this way. For example, an IC could be a called a distributed system if we considered transistors to be the nodes. One might then be tempted to use dimensional criteria, such as size or distance, to describe a distributed system. However, this, too, is not a relevant characteristic of a distributed system.

We now proceed to study different characteristics of distributed systems. We shall see that these aspects are not necessary conditions for a system to be considered distributed. Rather they should be seen as general indications that the system with these properties is likely to be a distributed one.

1. **Multiple processing nodes**.

2. **Costly communication**. In a distributed system, communication overheads are generally not assumed to be negligible, and we have to pay attention to them. How much, quantitatively, is considered to be "costly" is, of course, a matter of degree. However, if communication is seldom a matter of concern, then the system is not likely a distributed one. This aspect is not a requirement for distributed systems, but gives a strong hint that the system might be a distributed one.

   One example of a distributed system in which communication overheads have a dominant effect is multiprocessors. A major effort in designing such systems is spent in trying to reduce the effect of communication. Such issues of communication shall be studied in a more quantitative manner later on (*e.g.*, bandwidth and latency considerations).

3. **Tolerance of partial failures**. We expect a distributed system to go on working even if parts of it are broken. As before with communication, this is not an absolute requirement of distributed systems. In fact, fault tolerance is not usually that important an aspect in some multiprocessors which often have simple strategies for dealing with this issue.

   However many other systems do have very strong requirements for fault tolerance. In the case of the Internet, for example, people expect not to be affected by the failure of other machines. The aim of fault tolerance is to get high availibility. This is often achieved by replication. Fault tolerance is a fundamental part of the study of distributed systems, since to get fault tolerance in an efficient manner often requires a bit of care.

   (Lamport has offered the definition that a system is distributed when "I can't get my work done because some machine I have never heard of has failed.")

4. **Autonomy**. This is not a technical issue, but still a relevant characteristic of a distributed system. Often, all the nodes of a distributed system are not under the same management. Thus we are limited in our ability to apply a system-wide policy. The strongest example of such a system is the Internet.

   Once again, it should be noted that this is not an absolute distinguishing factor of distributed systems. For example, the nodes of a multiprocessor are usually *not* autonomous. For instance, in the CM-5 there is little autonomy since a small number of programmers determine which nodes would do what.

Another point to be noted is that a distributed system can be recursively implemented. A distributed system consists of nodes, but each of the nodes itself could also be a distributed system. There are other configurations possible. For example, a network of workstations is a distributed system, but each workstation itself is not usually considered a distributed system. Or, one may have a system in which the nodes are distributed systems, but the overall system is run synchronously, not as a distributed system.

Distributed systems also have some incidental (not so fundamental) characteristics which are described below. They also tend to be found in many systems which we would not consider to be distributed.

5. **Concurrency**. There are two clear reasons for studying concurrency. First, we need concurrency to make our systems go fast. Second, we often find that the distributed system under study requires fault tolerance and autonomy. In such a situation, the nodes *must* be running concurrently.

   Of course, there are always ways of stamping out concurrency, but this is often very difficult to do. Thus concurrency plays a major role in our understanding of distributed systems.

6. **Size/Scalability**. It is possible for a system to grow large, if one accepts the axioms of distributed systems. We should be able to "plug in" more stuff like nodes without changing the characteristics of the systems. The telephone network, and the email distribution system are examples of big systems that go into the tens of millions of nodes; Internet started as only a few machines, and now encompasses millions of nodes. One can say that big systems are usually distributed. On the other hand, simply buying more disks or memory does not in itself make a system distributed.

7. **Heterogeneity**. Many distributed systems have different kinds of nodes, for example the different workstations in a network of workstations. From a purist's point of view, this is not a matter of concern—it should not matter that one machine is Unix-based and another is DOS-based.

   However, in real and practical terms, we need to define certain specifications to which the different nodes have to adhere. Often it is very difficult to determine whether different nodes meet certain specifications. Some people might want to do away with specifications completely. However the resultant design would not be a system! There has to be some agreement and some interfaces for a design to be called a system.

   Another point of note here is that, since nodes may have different behaviors and may need different representations of information, if the system is properly specified then it is always possible to include a strange node into a distributed system by simply building an interface adaptor box.

We shall now look a little more at the specifics of distributed systems by taking apart one such system, and looking at each layer of abstraction in it. A good example is the ISO Reference Model (also known as the seven-layer cake!).

3 ISO Reference Model

The ISO Reference Model tells a story about how to think about communication in a distributed system. One must be careful not to take this model too literally. This model defines 7 layers which are mentioned in brief below.

1. **Physical**. From volts to bit sequences.
2. **Data Link**. From bit sequences to point-to-point messages.
3. **Network**. From point-to-point messages to routing of addressed messages.
4. **Transport**. Reliable messages and acks.
5. **Session**. Issues of naming and RPC.
6. **Presentation**. Murky relationship between data format and user.
7. **Application**. Not really relevant to distributed systems. (Pretty much a grab-bag for everything that isn't covered in 1–6.

The first four layers, Physical through Transport, are fairly standard and are typically needed in most systems. The next three layers are described more vaguely, and their existence is a little more controversial.

We shall now describe these layers in a little more detail. The last two layers, Presentation and Application, are murky in their detail and have little relevance to distributed systems, and hence will not be studied in any great detail.

4 General Principles

Before we start looking at various layers, it is useful to consider some general principles which will come up in a number of different circumstances.

1. **Recursion**. Characteristic abstractions at each level. We find that it is usually possible to use the implementation of a higher level for that of the lower level. This usually involves encapsulation, where we wrap messages inside each other. In another case, we multiplex several communication lines into one. In both cases, we can not tell what is going on without knowing both the *network* and the *context*.

   For example, we can have a single physical link but want to run several protocols (say, TCP and SNA) over it. There are two possible solutions:

   (a) we multiplex and tack a piece at the front of the message to identify its type, or

   (b) we simulate SNA on top of TCP by encapsulating the SNA message using TCP as the data link.

2. **Broadcast vs. Switch**. More on this topic will be discussed later but the basic ideas are represented in the Ethernet bus (a broadcast network), and the telephone system.

3. **Addressing and routing**. Addressing deals with where a message is going, and routing tries to get it there.

4. **Fault Tolerance**. An application which is at a higher level often lets the lower level protocols it sits on provide the fault tolerance (finding a way to get the message through). Redundancy is the most obvious way of achieving this. We can have redundancy in space by having more than one copy, or redundancy in time by retrying. (If transient errors are more likely we can just wait and retry. This is much cheaper than having more hardware and models the common failure mode of most network communications.)

5. **End-to-end reliability**. We consider two nodes trying to accomplish a joint computation (goal). There are two basic approaches for making this happen. In the first approach, we break the problem into little steps which concatenated together perform a single large behavior; this is like programming. There are two major problems with this technique:

   - We have to characterize the behavior correctly.
   - In chaining these little steps together, we have to consider all the possible ways in which each step can fail. This is too hard since there are often large numbers of them.

   Because of these drawbacks associated with chaining, we often use a second approach: we do the entire computation and perform a simple check on its result. To understand this second approach more fully, we will consider the task of copying a file. First, we do the copy with whatever flaky method we chose. Then we compute a checksum remotely and compare it to the local copy. If they don't agree, we try copying the file again. Checksums have the property that we can always meet any error rate that we find acceptable. This approach is great if the probability of failure is low. If the probability of failure is high, we can break the task into smaller pieces—this is more complex but much faster in face of high failure rates. For example, with a one percent failure rate ($p_{fail} = .01$) the cost is:

   $$1 + p_{fail}(1) + p_{fail}^2(1) + \cdots = 1 + .01 + .0001 + \cdots \approx 1.01$$

   So the cost is 1.01 times as long as if it didn't fail at all. This is fast path performance analysis (from Lecture 6) where the cost is

   $$1 + s \cdot m$$

   (where s is the slow path time and m is the miss rate of the fast path).

   For example, using main memory as disk cache: $1 + (10^4)(10^{-3}) = 11$ times as long as main memory access time. There are a number of games that must be played with balancing retry in End-to-End systems. We need some way of deciding when to retry. At high levels we expect a reply at some point. At low levels if we send a message out and we don't get one back, we don't know how long it is going to take to get one back (when we should retry), and in fact we may retry forever. The problem is that if we reply frequently, we increase the load, which implies lower performance, which implies more retries; this “death spiral” will lead to a dead spot. A possible solution is to use exponential backoff which doesn’t limit the maximum delay, although in some cases this may not be enough.

6. **Performance**. There are two performance questions we usually ask about distributed systems—“when we will get it” or bandwidth and latency, and “if we will get it” or congestion and availability. Better bandwidth and latency are always a good idea. They tend to have different names depending on the application area.

5 Physical/Data Link

This layer deals with taking a physical phenomenon and turning it into a stream of bits or symbols. The receiver sees changing voltages on a wire. We want to sample these changes at the middle of each signal segment. The problem is that we need some way of synchronizing; this requires knowing the frequency (how often) and the phase (where they start) of the signal. A standard solution is to introduce a marker signal which the receiver can follow for some time afterwards. The RS232 model is a system which uses just such a technique. The RS232 standard gives rise-time and fall-time specifications, and

Figure 2: Bit stream

also some coding scheme. The bit stream is as follows: there is a start bit, and then eight data bits, and then a stop bit. Thus, RS232 defines a clock edge at which we must sample the data—when see more than ten zeros then the next rise edge is the start bit. We get the frequency by knowing the approximate rate of the sender. This will work for a number of bits. For example if we guess time between bits is 100 microseconds when it is really 90 you would only be off by ten percent in the stop bit, and thus still get the correct answer.

Ethernet uses cell-encoding. In this method, both zeros and ones are signals; however, ones only last for half of the cycle. We then build a phase-lock loop circuit to lock into the signal. One problem with this solution is that the peak bandwidth is twice the signal frequency. Also, there is a problem with high frequencies which can cause attenuation of the cable.

To solve these problems, we combine the techniques of cell-encoding and having sync markers. This is done by having one of the bit patterns represent a sync marker and still not limit what can be data we encode the data before sending. FDDI uses what is called a 4-5 code (5-bits encode, 4-bits of data). We then send six sync codes in a row to guarentee a lock. We use other out-of-band codes to show start and end points. However, one drawback is that we give up 20% of the bandwidth in this scheme.

Yet another method of inserting synchs is to scramble the data by XORing it with a pseudo-random sequence. The synch is inserted in the scrambled datastream after every n bits. The receiver guesses the position of the synch and checks if it repeats after every n bits. Since the data is scrambled, it is unlikely that it will contain a similar periodic pattern that may be confused with the real synch. (The scrambling also avoids long sequences of zero's, which can cause problems for some detection methods.)

This scheme is used by the telephone company in T1 trunk lines. The T1 system uses a single synch bit after 24 bytes of scrambled data. There are 24 64-Kbit channels in a single T1 channel (thus there are 193 bits in a frame). The receiver makes a guess about the synch bit and checks if it repeats after 24 bytes. If not, the next bit position is checked, and so on. The advantage of the scheme is that only one bit is wasted per 24 bytes of data. This is much better than the 20% lost in the computer science style solution.

We now have messages. In FDDI there are markers that show the start and end. This is a bit harder in T1 if we don't want 192 bit messages, but that is the price paid for improved space utilization.

6 Networks

Networks are complex, and there are many forms. However, one main concept which must be resloved is: how to get from messages over a *single link* to a network. We need some form of switching so that a message can come in on one of a number links to a node, and then go out. To visualize this, we make a graph connecting such nodes. We can then use a grid pattern to go from a fixed fan-in, fan-out node to arbritrary sized virtual switching.

Offen we view this node (switch) as having an input part (a multiplexor, or mux) linked to an output part (demultiplexor, or demux). This view is sometimes lost in the implementation. In Ethernet we use a single wire for all traffic, and exponential backoff is used to handle collisions in the mux part of this implementation. We demux by the fact that each host connector only accepts broadcast packets with its address as recipient. A second common implementation of networks is the Token Ring, which is a point-to-point network. Once again we demux by address. The token is the multiplexor since a node must grab it before it can put a message on the ring. Overall, when considering the workings of a network there are three main axes to consider:

- Arbitration (chosing among senders)
- Addressing
- Availability

Failures in Ethernet do not cause problems unless the node just sends packets at random without looking at the ether. But Ethernet depends on everyone competing fairly. If a node is not fair and competes "aggressively", then it will *always* be able to "win"; it will always be able to put its messages on the network (assuming that everyone else is "fair". For this reason, Ethernet must have everyone competing fairly. Similarly, if a node fails in a token ring it will block further traffic. There are a number of solutions to these problems which will be discussed in a future lecture, along with more detail regarding the functioning of networks.