Distributed and Parallel Systems

We have discussed the abstractions and implementations that make up an individual computer system in considerable detail, and we've talked about how networks enable processes running on individual computers (hosts) to communicate. Communicating processes are the basis of a whole new set of abstractions that try to make it easier to program collections of hosts for greatly enhanced performance and reliability. A good analogy is that communicating processes are the assembly language of distributed and parallel systems.

This handout tries to summarize some of the important ideas at these higher levels. It is by no means exhaustive. Furthermore, there are many open areas of research at these levels. Perhaps this research will make distributed and parallel programming as easy as programming an individual machine.

Why Distributed and Parallel Programs?

There are generally three drivers behind using multiple hosts. First, some applications simple require more resources than are available on a single machine. For example, a web search engine like Google gets so much traffic that there simply aren't enough cycles on a single machine. Spreading the work out across multiple machines (100s in the case of Google) makes such application possible. The second driver is performance. Throwing more hosts at a problem, a scientific study or simulation such as SETI@Home, can get it done much faster than using a single machine. At its peak, SETI@Home exploited hundreds of thousands of machines. The third driver is reliability. Let's say you have a critical service. Suppose you map it to a single host and that host has a 0.1 chance of failing in a year. If it takes a day to recover, then you have an expected downtime of 2.4 hours every year. If you instead replicate the service over 10 hosts, the chance of all of them failing is 0.1^10 and your expected downtime is in the microsecond range.

Distributed and Parallel Algorithms

Although we haven't talked much about algorithms in this course, it is important to point out that the design of distributed or parallel algorithms is a bit different from their sequential counterparts. Distributed algorithms are designed to accomplish their work despite failures of hosts and network links, all without putting undo amounts of traffic on the network. On the other hand, parallel algorithm design usually assumes that failures are not an issue. Their design is strongly concerned with worst-case asymptotic performance. However, unlike sequential algorithms, there are two "big O" values of concern. One is "work complexity" – the amount of work that is done. The other is "depth complexity" – the longest path in the computation, or how long the algorithm takes given an infinite number of processors. For example, a parallel quick sort is of work complexity O(n log n), but it has depth complexity of only O(log n)!

Higher-level Communication Abstractions

Socket programming is too low level for many purposes. It's complex to set up a communication channel, there is no help in starting remote processes, communication is byte-stream oriented, and only communication between two individual hosts is a part of the model. However, we can build more sophisticated communication abstractions on top of sockets.

The simplest abstraction is that of **message passing**. Systems like PVM and MPI provide support for starting remote processes and communicating simply with messages instead of byte streams. Beyond simple host-to-host communication, these systems also implement **collective communication**. What this means is that you can express a **communication pattern** involving all of the hosts, and the system will optimize it. For example, the **all-to-all pattern**, in which, en masse, each host sends a message to every other host, is important in many parallel algorithms, but it is very difficult to schedule optimally on even a simple network. By making the pattern explicit instead of writing the message sends and receives directly, the programmer gives the message passing system the critical information it needs to even begin to do this scheduling.

A powerful abstraction in distributed systems is the **remote procedure call** or **RPC**. In RPC, we use a special **interface definition language** (**IDL) compiler** (also called a stub generator) to generate wrappers (stubs) that interface a regular procedure to the RPC run-time system. By using these wrappers and the RPC run-time, you can export any function you write so that it can be called by any host on the network. The caller uses another wrapper which makes it look like he's calling your function locally. In effect, he links with a wrapper that does the hard work of finding your function and calling it over the network, dealing with different endianness, alignment requirements, etc. Your wrapper does the hard work of making your function callable over the network. RPC technology dates to the early 80s, but it is continually reinvented. The latest commercial implementations are CORBA, Microsoft DCOM, and Java RMI. These systems are called **distributed object systems**, because they extend the RPC idea to objects. CORBA also allows you to call an object written in any language from some other language, solving some of the linking problems that we discussed.

Distributed shared memory or **DSM** extends the idea of communication via shared memory regions to the network, allowing regions to be shared between processes running on different hosts. However, this is not as simple as it seems because networks are (still) considerably slower than memory systems. Furthermore, if the hosts are **heterogenous** (different kinds of architectures and/or OSes), a simple extension of the intra-host shared memory model is not possible. Typically, DSM systems require that the programmer create shared blocks of data, carefully defining the type of data in the block so that the DSM system can translate between different kinds of hosts. Another issue is the granularity of updates to these shared blocks and the degree of consistency that is needed between different hosts' views of them. We discuss these **consistency model** issues further below because they apply in other distributed system contexts as well.

Languages for Distributed and Parallel Computing

One of the things that makes a single computer system so eminently programmable is the existence of programming languages that raise the level of abstraction at which one programs considerably. Furthermore, the compiler toolchains that implement these languages hide many details from us. Even C, which is about as primitive of a programming language as there is, is a huge step up from assembly language programming. Other languages, such as C++, Java, Perl, Python, Rexx, Matlab, Fortran 9X, Lisp, Scheme, and ML, raise these abstractions much further. ML programs, for example, are mathematical objects about which a compiler can reason using logic.

Sadly, the state of the art in languages for distributed and parallel computing is much less refined. Over the past 25+ years, massive amounts of research dollars have been spent with little success in pursuit of **automatic parallelization** – producing parallel and distributed programs by compile-time analysis of programs written in ordinary sequential languages like Fortran. The major success story here is in **automatic vectorization**, which is particular to vector machines (the registers hold vectors and the instructions operate on vectors). While vector machine technology was originally confined to very expensive supercomputers, it has slowly migrated to the desktop. Intel's MMX and IBM/Motorola's Altivec are simple implementations of vector processing on mainstream processors.

There has been much greater success in developing **explicitly parallel programming languages** and compilers that support them. However, the target for these languages has largely been the scientific community and the parallel algorithms community. These languages allow the programmer to specifiy collections of objects and explicitly operations on these collections. The compiler and run-time distribute these collections across hosts and implement parallel operations on them as sequential operations and message passing. In the case of High Performance Fortran (HPF) or Parallel Matlab, the collections are arrays and the operations are operations on whole arrays, vectors, or slices of either. Other collections are possible. For example, the Nesl language supports arbitrary nested lists of arbitrary objects. A parallelizing compiler for one of these languages will typically translate a high-level, explicitly parallel program down to a sequential program the includes message passing. For example, HPF might compile to Fortran (or C) with message passing calls to MPI.

The state of languages for distributed computing is still quite immature. One bright spot is **interface definition languages** and their compilers, which form the backbone of RPC systems. An IDL lets you define the interface of an object or function without specifying how it is implemented. The compiler can then generate code that allows that object to be used in many contexts, including across a network. A common example is CORBA IDL.

The Consensus Problem and Consistency Models

Many issues in distributed computing ultimately boil down to either the consensus problem. Since we have multiple machines, we will often want to replicate an object, perhaps having one copy of the object per machine. The consensus problem is how to keep those copies consistent with each other. At a high level, this seems like an either-or

propostion: either two replicas are the same or they are not. However, the known approaches for forcing replicas to be exactly the same at all times in the face of independent updates happening on the different processors are simply too slow to be used in practice – they ameliorate the very benefits that we expect to get from having multiple machines.

To make progress, researchers have defined different forms of consistency. One we might refer to as "**consensus consistency**". The idea here is that the replicas are consistent if we can query some small subset of them for their values and combine their answers to produce the "actual" value. Another family of consistency models comes from considering how updates to a replica on one processor are perceived on another processor. Your book described one model here, the **sequential consistency** model, in the context of thread programming. The idea of sequential consistency and its family apply at many levels of computer systems. The idea here is that the observing processor sees the updates in precisely the order in which they were issued. For various reasons, this model is very difficult to implement efficiently. Hence, "looser" consistency models have been developed. For example, **release consistency** introduces specific updates, often called **barriers**. The updates between barriers are perceived in any order, but the barriers are totally ordered. Hence, when the observing processor sees the barrier, it knows it has seen all the updates prior to the barrier. DSM systems such as Treadmarks rely on even more relaxed forms of consistency in order to function.

Another approach to consistency is based effectively on barriers with respect to time instead of with respect to updates. This is known as **virtual synchrony**. ISIS and HORUS are the best known implementations.

Consensus problems occur frequently in the Internet, because many services, such as DNS, web caches such as Akamai and Squid, and netnews, rely on caching to provide high performance. Consensus in distributed systems is a deep, intellectually fascinating, area of work.