Lectures on distributed systems:

A taxonomy of distributed systems

Paul Krzyzanowski

Introduction

DISTRIBUTED SYSTEMS APPEARED relatively recently in the brief history of computer systems. Several factors contributed to this. Computers got smaller and cheaper: we can fit more of them in a given space and we can afford to do so. Tens to thousands can fit in a box whereas in the past only one would fit in a good-sized room. Their price often ranges from less than ten to a few thousand dollars instead of several million dollars. More importantly, computers are faster. Network communication takes computational effort. A slower computer would spend a greater fraction of its time working on communicating rather than working on the user's program. Couple this with past CPU performance and cost and networking just wasn't viable. Finally, interconnect technologies have advanced to the point where it is very easy and inexpensive to connect computers together. Over local area networks, we can expect connectivity in the range of tens of Mbits/sec to a Gbit/sec.

Tanenbaum defines a distributed system as a "*collection of independent computers that appear to the users of the system as a single computer.*" There are two essential points in this definition. The first is the use of the word *independent*. This means that, architecturally, the machines are capable of operating independently. The second point is that the software enables this set of connected machines to appear as a *single computer* to the users of the system. This is known as the **single system image** and is a major goal in designing distributed systems that are easy to maintain and operate.

Why build them?

Just because it is easy and inexpensive to connect multiple computers together does not necessarily mean that it is a good idea to do so. There are genuine benefits in building distributed systems:

Price/performance ratio. You don't get twice the performance for twice the price in buying computers. Processors are only so fast and the price/performance curve becomes nonlinear and steep very quickly. With multiple CPUs, we can get (almost) double the performance for double the money (as long as we can figure out how to keep the processors busy and the overhead negligible).

Distributing machines may make sense. It makes sense to put the CPUs for ATM cash machines at the source, each networked with the bank. Each bank can have one or

more computers networked with each other and with other banks. For computer graphics, it makes sense to put the graphics processing at the user's terminal to maximize the bandwidth between the device and processor.

Computer supported cooperative networking. Users that are geographically separated can now work and play together. Examples of this are electronic whiteboards, distributed document systems, audio/video teleconferencing, email, file transfer, and games such as Doom, Quake, Age of Empires, and Duke Nuke'em, Starcraft, and scores of others.

Increased reliability. If a small percentage of machines break, the rest of the system remains intact and can do useful work.

Incremental growth. A company may buy a computer. Eventually the workload is too great for the machine. The only option is to replace the computer with a faster one. Networking allows you to add on to an existing infrastructure.

Remote services. Users may need to access information held by others at their systems. Examples of this include web browsing, remote file access, and programs such as Napster and Gnutella to access MP3 music.

Mobility. Users move around with their laptop computers, Palm Pilots, and WAP phones. It is not feasible for them to carry all the information they need with them.

A distributed system has distinct advantages over a set of non-networked smaller computers. Data can be shared dynamically – giving private copies (via floppy disk, for example) does not work if the data is changing. Peripherals can also be shared. Some peripherals are expensive and/or infrequently used so it is not justifiable to give each PC a peripheral. These peripherals include optical and tape jukeboxes, typesetters, large format color printers and expensive drum scanners. Machines themselves can be shared and workload can be distributed amongst idle machines. Finally, networked machines are useful for supporting person-to-person networking: exchanging email, file transfer, and information access (e.g., the web). As desirable as they may now be, distributed systems are not without problems:

- Designing, implementing and using distributed software may be difficult. Issues of creating operating systems and/or languages that support distributed systems arise.
- The network may lose messages and/or become overloaded. Rewiring the network can be costly and difficult.
- Security becomes a far greater concern. Easy and convenient data access from anywhere creates security problems.

Interconnect

There are different ways in which we can connect CPUs together. The most widely used classification scheme (**taxonomy**) is that created by Flynn in 1972. It classifies machines by the number of instruction streams and the number of data streams. An instruction stream

refers to the sequence of instructions that the computer processes. Multiple instruction streams means that different instructions can be executed concurrently. Data streams refer to memory operations. Four combinations are possible:

SISD Single instruction stream, single data stream. This is the traditional uniprocessor computer.

SIMD Single instruction stream, multiple data streams. This is an array processor; a single instruction operates on many data units in parallel.

MISD Having multiple concurrent instructions operating on a single data element makes no sense. This isn't a useful category.

MIMD Multiple instruction stream, multiple data streams. This is a broad category covering all forms of machines that contain multiple computers, each with a program counter, program, and data. It covers parallel and distributed systems.

Since the MIMD category is of particular interest to us, we can divide it into further classifications. Three areas are of interest to us:

Memory

We refer to machines with shared memory as **multiprocessors** and to machines without shared memory as **multicomputers**. A multiprocessor contains a single virtual address space. If one processor writes to a memory location, we expect another processor to read the value from that same location. A multicomputer is a system in which each machine has its own memory and address space.

Interconnection network

Machines can be connected by either a *bus* or a *switched* network. On a bus, a single network, bus, or cable connects all machines. The bandwidth on the interconnection is shared. On a switched network, individual connections exist between machines, guaranteeing the full available bandwidth between machines.

Coupling

A **tightly-coupled** system is one where the components tend to be reliably connected in close proximity. It is characterized by short message delays, high bandwidth, and high total system reliability. A **loosely-coupled** system is one where the components tend to be distributed. Message delays tend to be longer and bandwidth tends to be lower than in closely-coupled systems. Reliability expectations are that individual components may fail without affecting the functionality of other components.

Bus-based multiprocessors

In a bus-based system, all CPUs are connected to one bus (Figure 1). System memory and peripherals are also connected to that bus. If CPU *A* writes a word to memory and CPU *B* can read that word back immediately, the memory is **coherent**.

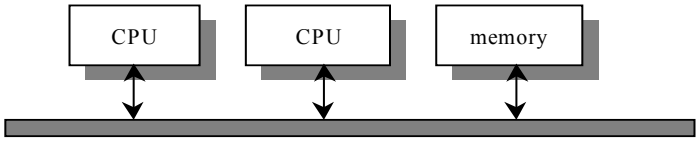

Figure 1. Bus-based interconnect

A bus can get overloaded rather quickly with each CPU accessing the bus for all data and instructions. A solution to this is to add **cache memory** between the CPU and the bus (Figure 2). The cache holds the most recently accessed regions of memory. This way, the CPU only has to go out to the bus to access main memory only when the regions are not in its cache.

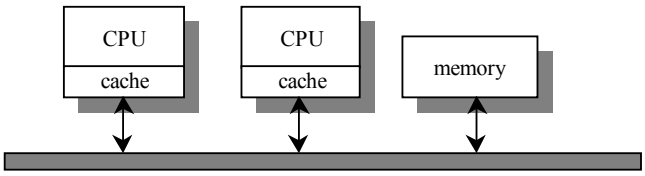

Figure 2. Bus-based interconnect with cache

The problem that arises now is that if two CPUs access the same word (or same region of memory) they load it into their respective caches and make future references from their cache. Suppose CPU *A* modifies a memory location. The modification is local to its cache so when CPU *B* reads that memory location, it will not get *A*'s modification. One solution to this is to use a **write-through cache**. In this case, any write is written not only to the cache, but also sent on the bus to main memory. Writes generate bus traffic now, but reads generate it only if the data needed is not cached. We expect systems to have far more reads than writes.

This alone is not sufficient, since other CPU caches may still store local copies of data that has now been modified. We can solve this by having every cache monitor the bus. If a cache sees a write to a memory location that it has cached, it either removes the entry in its cache (invalidates it) or updates it with the new data that's on the bus[1]. If it ever needs that region of memory again, it will have to load it from main memory. This is known as a **snoopy cache** (because it *snoops* on the bus).

[1] This is a design choice. Either way, memory coherence is achieved.

Switched multiprocessors

A bus-based architecture doesn't scale to a large number of CPUs (e.g. 64). Using switches enables us to achieve a far greater CPU density in multiprocessor systems. An $m \times n$ **crossbar switch** is a switch that allows any of m elements to be switched to any of n elements. A crossbar switch contains a *crosspoint* switch at each switching point in the $m \times n$ array, so $m \times n$ crosspoint switches are needed (Figure 3). To use a crossbar switch, we place the CPUs on one axis (e.g. m) and the break the memory into a number of chunks which are placed on the second axis (e.g. n memory chunks). There will be a delay only when multiple CPUs try to access the same memory group.

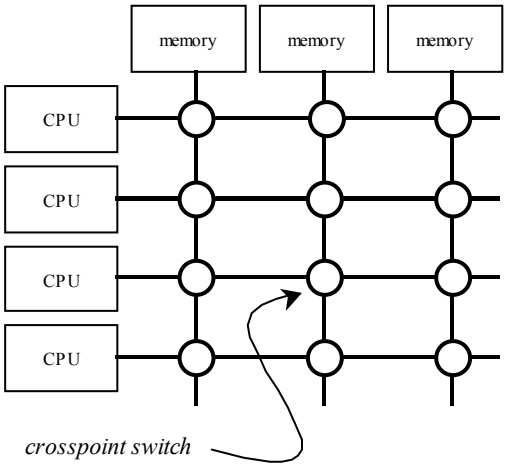

Figure 3. Crossbar interconnect

A problem with crossbar switches is that they are expensive: to connect n CPUs with n memory modules requires n^2 crosspoint switches. We'd like an alternative to using this many switches. To reduce the number of switches and maintain the same connectivity requires increasing the number of switching stages. This results in an **omega** network (Figure 4), which, for a system of n CPUs and n memory modules, requires $\log n$ (base 2) switching stages, each with $n/2$ switches for a total of $(n \log n)/2$ switches. This is better than n^2 but can still amount to many switches. As we add more switching stages, we find that our delay increases. With 1024 CPUs and memories, we have to pass through ten switching stages to get to the memory and through ten to get back.

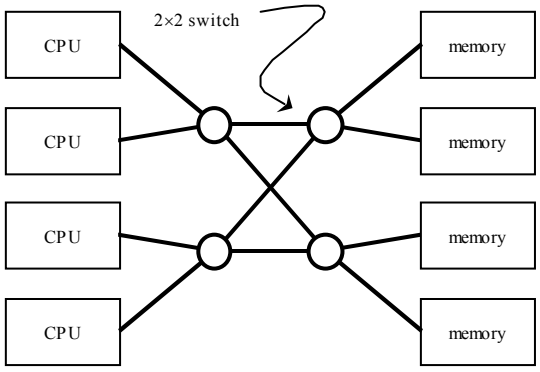

Figure 4. Omega interconnect

To try to avoid these delays, we can use a hierarchical memory access system: each CPU can access its own memory quickly but accessing other CPU's memory takes longer. This is known as a **Non-Uniform Memory Access**, or **NUMA**, architecture. It provides better average access time but placement of code and data to optimize performance becomes difficult.

Bus-based multicomputers

Bus-based multicomputers are easier to design in that we don't need to contend with issues of shared memory: every CPU simply has its own local memory. However, without

shared memory, some other communication mechanism is needed so that processes can communicate and synchronize as needed. The communication network between the two is a bus (for example, an Ethernet local area network). The traffic requirements are typically far lower than those for memory access (so more systems can be attached to the bus). A bus can either be a system bus or a local area network[2]. Bus-based multicomputers most commonly manifest themselves as a collection of workstations on a local area network.

Switched multicomputers

In a switched multicomputer system, each CPU still has its own local memory but it also has a switched interconnect to its neighbors. Common arrangements are a grid, cube, or hypercube network. Only nearest neighbors are connected in this network; messages to others require multiple hops.

Software issues

The software design goal in building a distributed system is to create a **Single System Image** - have a collection of independent computers appear as a single system to the user(s). By single system, we refer to creating a system in which the user is not aware of the presence of multiple computers or of distribution.

In discussing software for distributed systems, it makes sense to distinguish loosely-coupled vs. tightly-coupled software. While this is a continuum without demarcation, by **loosely-coupled** we refer to software in which the systems interact with each other to a limited extent as needed. For the most part, they operate as fully-functioning stand-alone machines. If the network goes down, things are pretty much functional. Loosely coupled systems may be ones in which there are shared devices or services (parts of file service, web service). With **tightly-coupled** software, there is a strong dependence on other machines for all aspects of the system. Essentially, both the interconnect and functioning of the remote systems are necessary for the local system's operation.

The most common distributed systems today are those with loosely-coupled software and loosely coupled hardware. The quintessential example is that of workstations (each with its own CPU and operating system) on a LAN. Interaction is often primitive explicit interaction, with programs such as `rcp` and `rlogin`. File servers may also be present, which accept requests for files and provide the data. There is a high degree of autonomy and few system-wide requirements.

The next step in building distributed systems is placing tightly-coupled software on loosely-coupled hardware. With this structure we attempt to make a network of machines

[2] System busses generally have speeds of about 300 Mbps – 1 Gbps. Typical speeds for local area networks are from 10 Mbps to 1 Gbps, with some operating in the Kbps and low megabit per seconds range (for infrared and wireless).

appear as one single timesharing system, realizing the single system image. Users should not be aware of the fact that the machine is distributed and contains multiple CPUs. If we succeed in this, we will have a true distributed system. To accomplish this, we need certain capabilities:

- A single global IPC mechanism (any process should be able to talk to any other process in the same manner, whether it's local or remote).
- A global protection scheme.
- Uniform naming from anywhere; the file system should look the same.
- Same system call interface everywhere.

The kernel on each machine is responsible for controlling its own resources (such as doing its own memory management/paging).

Multiprocessor time-sharing systems employing tightly-coupled hardware and software are rather common. Since memory is shared, all operating system structures can be shared. In fact, as long as critical sections are properly taken care of, a traditional uniprocessor system does not need a great deal of modification. A single run queue is employed amongst all the processors. When a CPU is ready to call the scheduler, it accesses the single run queue (exclusively, of course). The file system interface can remain as is (with a shared buffer cache) as can the system call interface (traps).

Design issues

There are a number of issues with which a designer of a distributed system has to contend. Tanenbaum enumerates them:

Transparency
At the high levels, transparency means hiding distribution from the users. At the low levels, transparency means hiding the distribution from the programs. There are several forms of transparency:

Location transparency
Users don't care where the resources are located.

Migration transparency
Resources may move at will.

Replication transparency
Users cannot tell whether there are multiple copies of the same resource.

Concurrency transparency
Users share resources transparently with each other without interference.

Parallelism transparency
Operations can take place in parallel without the users knowing.

Flexibility
It should be easy to develop distributed systems. One popular approach is through the use of a **microkernel**. A microkernel is a departure from the monolithic operating systems that try to handle all system requests. Instead, it supports only the very basic operations: IPC, some memory management, a small

amount of process management, and low-level I/O. All else is performed by user-level servers.

Reliability
We strive for building highly reliable and highly available systems. **Availability** is the fraction of time that a system is usable. We can achieve it through redundancy and not requiring the simultaneous functioning of a large number of components. **Reliability** encompasses a few factors: data must not get lost, the system must be secure, and the system must be fault tolerant.

Performance
We have to understand the environment in which the system may operate. The communication links may be slow and affect network performance. If we exploit parallelism, it may be on a fine grain (within a procedure, array ops, etc.) or a coarse grain (procedure level, service level).

Scalability
We'd like a distributed system to scale indefinitely. This generally won't be possible, but the extent of scalability will always be a consideration. In evaluating algorithms, we'd like to consider distributable algorithms vs. centralized ones.

Service models

Computers can perform various functions and each unit in a distributed system may be responsible for only a set number of functions in an organization. We consider the concept of *service models* as a taxonomy of system configurations.

Centralized model

A **centralized model** (Figure 5) is one in which there is no networking. All aspects of the application are hosted on one machine and users directly connect to that machine. This is epitomized by the classic mainframe time-sharing system. The computer may contain one or more CPUs and users communicate with it via terminals that have a direct (e.g., serial) connection to it.

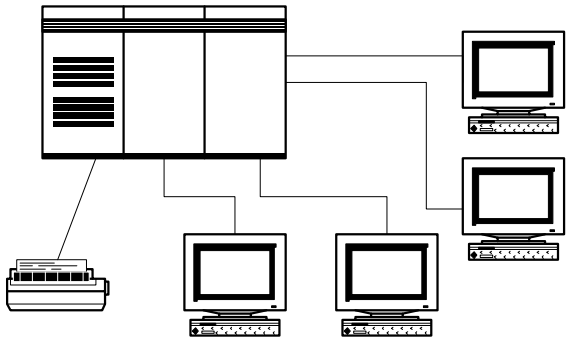

Figure 5. Centralized model

The main problem with the centralized model is that it is not easily scalable. There is a limit to the number of CPUs in a system and eventually the entire system needs to be upgraded or replaced. A centralized system has a problem of multiple entities contending for the same resource (e.g. CPUs for the system bus).

Client-server model

The **client-server model** (Figure 6) is a popular networked model consisting of three components. A *service* is the task that a particular machine can perform. For example, offering files over a network, the ability to execute certain commands, or routing data to a printer. A *server* is the machine that performs the task (the machine that hosts the service). A machine that is primarily recognized for the service it provides is often referred to as a print server, file server, et al. The *client* is a machine that is requesting the service. The labels *client* and *server* are within the context of a particular service; a client can also be a server.

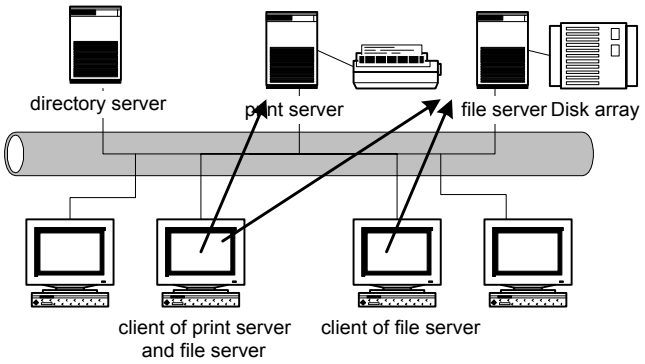

Figure 6. Client-server model

A particular case of the client-server model is the **workstation model**, where clients are generally computers that are used by one user at a time (e.g. a PC on a network).

Peer-to-peer model

The client-server model assumes that certain machines are better suited for providing certain services. For instance, a file server may be a system with a large amount of disk space and backup facilities. A **peer-to-peer model** (Figure 7) assumes that each machine has somewhat equivalent capabilities, that no machine is dedicated to serving others. An example of this is a collection of PCs in a small office or home. Networking allows people to access each other's files and send email but no machine is relegated to a specific set of services.

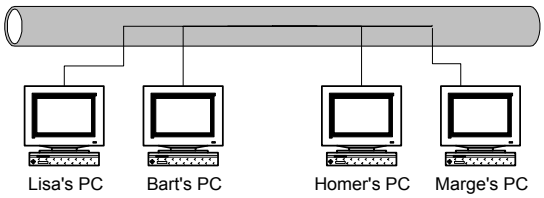

Figure 7. Peer-to-peer model

Thin and thick clients

We can further explore the client-server environment by considering the partitioning of software between the client and the server: what fraction of the task does the client process before giving the work to the server? There are two schools of design, identified as **thin client** and **thick client**.

A **thin client** is designed around the premise that the amount of client software should be small and the bulk of processing takes place on the servers. Initially, the term referred to only software partitioning, but because the software requirements are minimal, less hardware is needed to run the software. Now, thin client can also refer to a client computing machine that needs not be the best and fastest available technology to perform its task acceptably. Examples of thin clients are an X terminal (e.g., an NCD X

terminal, a PC dedicated to running exceed), a Network PC (NetPC, proposed by Intel and Microsoft), Oracle and Sun's Network Computer (NC), perhaps running a Java-based operating system, and systems like the *Netpliance*. With thin clients, there is no need for much administration, expansion slots, CDs, or even disks. The thin client can be considered to be an *information appliance* (wireless device, or set-top box) that only needs connectivity to resource-rich networking.

The opposite of a thin client is a **thick client** (or *fat client*). In this configuration, the client performs the bulk of data processing operations. A server may perform rather rudimentary tasks such as storing and retrieving data. Today's Microsoft/Intel-dominated PC world is an example of thick clients. Servers are useful (providing web service or file storage service), but the bulk of data processing generally takes place on the client (e.g. word processing, spreadsheets). This creates an ever-increasing need for faster processors (thanks to forever-bloating software), high capacity storage devices (thanks also to the bloatware), and a very significant amount of system configuration and administration). An argument for thin-clients is that work is offloaded from the clients, allowing users to treat their systems as appliances and not hassle with administrative aspects or constant upgrades. In defense of thick-clients, computers and related peripherals are becoming ever faster and cheaper. What is the point of off-loading computation on a server when the client is amply capable of performing it without burdening the server or forcing the user to deal with network latencies?

Multi-tier client-server architectures

For certain services, it may make sense to have a hierarchy of connectivity. For instance, a server, in performing its task, may contact a server of a different type. This leads us to examine *multi-tier* architectures. The traditional client-server architecture is a **two-tier architecture** (Figure 8). The user interface generally runs on a user's desktop and application services are provided by a server (for example, a database). In this architecture, performance often suffers with large user communities (e.g., hundreds). The server may end up spending too much time managing connections and serving static content and does not have enough cycles left to perform the needed work in a timely manner. In addition, certain services themselves may be performance hogs and contend for the now-precious CPU resource. Moreover, many legacy services (e.g., banking) may have to run on certain environments that may be poorly adapted to networked applications.

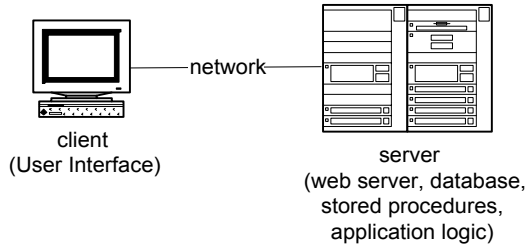

Figure 8. Two-tier architecture

These problems led to a popular design known as a **three-tier architecture** (Figure 9). Here, a middle tier is added between the client providing the user

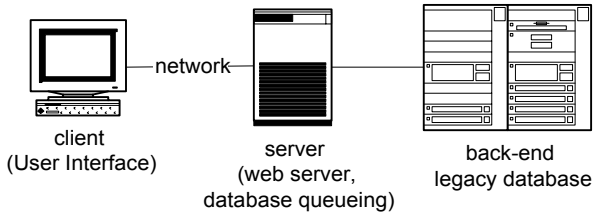

Figure 9. Three-tier architecture

interface and the application server. The middle tier can perform:

- queuing and scheduling of user requests
- connection management and format conversions
- application execution (with connections to a back-end database or legacy application)

It may also employ a Transaction Processor (TP) monitor to queue messages and schedule back-end database transactions. There is no need to stop at three tiers. Depending on the service to be provided, it may make sense to employ additional tiers. For example, a common infrastructure used in may of today's web sites has a web server (responsible for getting connections and serving static content) talking to an application server (running business logic implemented, for example, as java servlets), which in turn talks to a transaction processor that coordinates activity amongst a number of back-end databases.

Processor-pool model

One issue that has not been addressed thus far is that of idle workstations, or computing resources. One option is to simply ignore them: it is no longer a sin to let computers sit idle. Another option is to use all available computing resources for running jobs. In the most intelligent case, an operating system can automatically start processes on idle machines and even migrate processes to machines with the most available CPU cycles. In a less intelligent case, a user may be able to manually start or move processes on available systems. Going one step further, and realizing the low cost of processors, what if there was a lot of computing power available to the user (for example, dozens or hundreds of CPUs per user)? This leads us to the **processor pool model**. In this model, we maintain a collection of CPUs that can be dynamically assigned to processes on demand. This collection need not be the set of idle workstations. We can have racks of CPUs connected to a high-speed network with the end-users only having thin clients: machine suitable for managing input and output (X servers, graphical workstations, PCs).