Distributed Systems

Introduction

By Paul Krzyzanowski
October 4, 2018

Introduction

Commercially-available computers have been around since the early 1950s. Remote data communication has been around for about the same time if we count transmitting bytes via punched paper tape on a teletype that is connected to a phone. Indeed, for some of the early history of networking the phone network served as a data network, with digital signals modulated onto voice frequencies and transmitted via a modem; it was the only wide-area communication infrastructure we had. By the 1960s, however, real data communication networks were being deployed. What happened that made networked computers so increasingly interesting now than decades ago? The answer is the steady and rapid advancement of four related technologies.

1. *Networks*
   Data networks went from being rare to being completely ubiquitous. Networking hardware was being deployed as far back as the 1960s. In 1976, Robert Metcalfe presented the concept of the *ethernet*, a relatively low-cost packet-based, shared bus network that communicated over thick coaxial cable at 2.94 megabits per second[1]. By 1980, Ethernet became a de facto standard, running at 10 megabits per second. As adoption of the technology grew, it advanced on several fronts.

   - Speeds increased dramatically (10 Mbps in 1985, 10 Mbps on twisted pair wiring in 1991, 100 Mbps in 1995, 1 Gbps in 1998, 10 Gbps in 2001, and 40 and 100 Gbps between 2005 and 2009.
   - Cabling became cheaper and lighter. In 1985, twisted pair wiring at 1 Mbps became an alternative to using thick coax cable. In 1999, the 802.11b protocol was standardized, giving us Wi-Fi, essentially wireless ethernet (not quite the same, but close enough). Now, Wi-Fi is the norm, offering speeds up to several gigabits per second via technologies such as multiple data streams, beamforming antennae, and MU-MIMO.
   - Ethernet switches gave us scalable bandwidth. Early coax-based ethernet was a shared communication channel. If other computers on the network were generating a lot of traffic, your network performance suffered. By 1991, ethernet switches could move data between any pair of communicating devices without affecting the network congestion of other nodes on the network.

   All this contributed to vastly faster, cheaper, and scalable bandwidth, which made it quick and easy to send vast quantities of data between machines.

Equally, or even more important, is the emergence of high speed, low cost wide area network connectivity (outside the home or company). For wide area networking, modems over telephone lines were the dominant mass market means of connectivity throughout the 1990s, at which time data over cable television networks and high-speed data over non-voice frequencies on telephone lines (ADSL) began to be deployed. A $1,000 modem in 1988 transferred data at 2,400 bits per second. A 6 megabyte MP3 file would take over six hours to download. The same file takes just a couple of seconds to download via my cable modem. Streaming high definition video was inconceivable back then.

Back in 1985, there were only 1,961 hosts on the Internet. In 2018, there were over a billion. Robert Metcalfe posited a "law" that stated *The value of a telecommunications network is proportional to the square of the number of connected users of the system*. If you think of all machines on a network as nodes on a graph and each node can talk to every other node, forming an edge to every other node, we have a graph with an order n-squared number of edges. What this "law" points out is that networks, both physical and social, become a lot more interesting as more connections are possible. In the social realm, the "law" gives networks significant staying power. Facebook gives you the chance to connect with around a billion people without them having to register with the service because they are already on it. Even if you find competing services (e.g., Google+) more appealing and want to migrate away from Facebook, you have to convince your friends to use the service, many of whom will be reluctant since they have to do the same with their friends. Services such as eBay are useful because 175 million people may potentially see the junk you are trying to sell. You may launch a better and cheaper auction site, but people will hesitate joining because they will not have the user community to drive up bids.

2. *Processors*

   Computers got smaller, cheaper, faster, and more power efficient. We can fit more of them in a given space and we can afford to do so, both in terms of cost and power. Microprocessors became the leading drivers of advances in computer architecture. A room that used to be able to hold one large computer can now be filled with racks of tens of thousands of computers. Their price often ranges from less than ten to a few thousand dollars instead of several million dollars.

   Computers have been getting faster. Network communication takes computational effort. Data packets often need to have their checksums verified, contents decrypted, and data parsed. The data itself may need to be mpved among multiple buffers. A slow computer will spend a greater fraction of its time working on communicating rather than working on the user's program. Given the low performance and high cost of computers in the past, networking just was often not viable on many microprocessors; protocols such as TCP/IP were burdensome and protocols involving encryption and complex parsing could tax the processor to the point where it could perform no useful work.

3. *Memory*

   In 1977, a typical minicomputer had between 128 and 256 KB of memory. A typical microcomputer had between 16 and 64 KB. Average prices were around $32,000 per megabyte. By 1982, a megabyte of memory cost $8,000. A megabyte isn't enough to even store a compressed JPEG image from a modern digital camera, many of which produce raw files that are often between 50 and 75 MB. This doesn't even factor in the size of the operating system, programs, and memory buffers. By 2018, DRAM prices were around $8 per gigabyte, or around 0.6 cents per megabyte. Equipping a personal computer with several gigabytes of DRAM is often a negligible expense.

4. *Storage*
   A similar dramatic price shift took place with non-volatile storage as with memory. A personal computer in 1977 wouldn't have hard disks and one would shuffle 360 KB floppy disks in and out of the computer. A MITS Altair floppy disk kit cost $1,480 and disks cost $4.25 each. You'd need 2,713 of them, and over $11,000, to get a gigabyte of storage. In 2018, a 4 TB SATA disk drive can be purchased for under $100. Storing hundreds of movies, thousands of songs, and tens of thousands of photos is not even a strain on storage resources.

Interconnect

There are different ways in which we can connect processors together. The most widely used classification scheme (taxonomy) is that created by Michael J. Flynn in 1972 and known as *Flynn's taxonomy*. It classifies computer architectures by the number of concurrent 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 performed by instructions. Four combinations are possible:

Name	Description
SISD	Single instruction stream, single data stream. This is the traditional uniprocessor computer. There is no parallelism. A single sequence of instructions is processed with each instruction operating on single elements of data.
SIMD	Single instruction stream, multiple data streams. This is an array processor; a single instruction operates on many data units in parallel. A common example of this is the architecure of GPUs, graphics processing units. The Intel architecture also supports a set of instructions that operate on vectors, performing the same operations on multiple data objects. These are called *Streaming SIMD Extensions* (SSE) and *Advanced Vector Extensions* (AVX)[2].
MISD	Having multiple concurrent instructions operating on a single data element makes no sense. This isn't a useful category but is occasionally used to refer to fault-tolerant systems where multiple processors operate on the same data stream and compare the outcome to ensure that none are malfunctioning.
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 both parallel and distributed systems.

MIMD is the category that 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 physical address space. If one processor writes to a memory location, we expect another processor to be able to read that value from that same location. A multicomputer is a system in which each machine has its own memory and address space. This is the key distinction between **parallel** (**multiprocesssor**) and **distributed** (**multicomputer**) systems.

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.

Distributed Systems

When we think of what a distributed system may be, we often think of it as a bunch of computers that work together to provide some kind of service. In many cases, we think of a huge collection of servers at data centers, since we know that Google search is not handled by a single server. Neither are interactions with Amazon, Twitter, Facebook, or any large-scale service. This view of distributed systems is not wrong but distributed systems need not be large scale. A home file server is a part of a distributed system. So is a wireless Bluetooth keyboard that interacts with a laptop.

More formally, we define a **distributed system** as a *collection of independent, autonomous hosts connected through a communication network*. By *independent, autonomous hosts*, we refer to multiple stand-alone computers. Each runs as a separate system, booting its own operating system and running its own programs. None of these computers is dependent on any other one to function. Specifically, the computers in a distributed system have:

1. No shared memory.
2. No shared clock.

More often than not, as far as software is concerned, they also have no shared operating system, allowing each computer to function truly independently. In some cases, however, a distributed operating system may provide low-level services for starting processes on different computers, allowing processes on different computers to communicate easily, or migrating processes from one computer to another. Processes may also require the availability of certain remote resources, such as file servers. However, the case where processes expect services from remote computers does not take away from the fact that the computers themselves are independent.

In contrast to parallel systems, distributed systems (multicomputers, or networks of computers) are characterized by a lack of shared memory. Because processors do not share memory, we need to have a way for them to communicate. We use a communications *network* for this. Because they are a collection of autonomous machines, the failure of one system does not imply that other systems will fail. It is very conceivable that we can have a *partial failure*, where some system are working and others are not.

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 before processes exhibit performance degradation. Bus-based multicomputers most commonly manifest themselves as a collection of workstations on a local area network.

Switched multicomputers

A switched multicomputer system is one where we use a switched interconnect instead of a bus-based one. A bus-based connection requires that all hosts on the network share the communications bus, which results in increased congestion as more and more hosts are on the system. With a switched interconnect, all hosts connect to a network switch. The switch moves traffic only between communicating hosts, allowing other hosts to communicate without seeing their network speeds diminish. The huge benefit of switching is that it gives us a scalable network, where we can add more hosts without seeing a decrease in performance. The common example is of an ethernet switch. The original ethernet network was designed as a bus-based medium, where all hosts tapped into a shared coaxial cable (the bus). When ethernet moved from a coax cable to twisted pair wiring, each host was connected to a hub. The hub copied any traffic to all connected wires, making the cables look like one shared ethernet cable. Later, hubs evolved into switches, where the movement of traffic from one cable to another was handled more intelligently and data was not sent onto cables where the connected host does not need to see it.

Distributed Systems Software

Andrew Tanenbaum further 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 fits with our overall definition of distributed systems and means that the machines are architecturally separate, standalone, machines. 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 a *single system image* and is a major goal in designing distributed systems that are easy to maintain and operate.

There is no single definition or goal of distributed software but in designing distributed software, we often touch upon the same set of goals and problems. These general goals are *scalability*, *fault tolerance*, and *transparency*. Scalability means that we can design a system that can perform better as we add more systems to it. Fault tolerance means that the system can survive the failure of components and continue functioning as intended because we architected redundant components into the design. Transparency refers to hiding the fact that we may have many machines involved and scalability refers to creating software systems that can scale well as we add more machines because the number of users, data, or computations involved go from small to huge.

The general definition of a distributed system as a collection of independent, autonomous hosts connected through a communication network working together to perform a service covers problems as diverse as:

- A network of redundant web servers
- Thousands of machines participating together in processing your search query
- A fault tolerant shopping cart as you browse Amazon
- Thousands of machines rendering frames for an animated movie
- Light switches, thermostats, and switched outlets forming an ad hoc network in your home

One design goal in building a distributed system is to create a *single system image*; to 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. A simple example is that of workstations (each with its own CPU and operating system) on a LAN. The distribution is often primitive with explicit user interaction (no or minimal transparency) via programs such as *scp* and *ssh*. File servers may also be present, which accept requests for files and provide the data. There is a high degree of autonomy in these systems 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 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 distributed operating 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).

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.

Cooperative and social networking

Users that are geographically separated can now work and play together. Examples of this are plenty: distributed document systems, audio/video conferencing, email, multiplayer games, auctions, and social networks.

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 BitTorrent to retrieve large files.

Mobility

Users move around with their laptop computers and 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 computers. Data can be shared dynamically. Giving everyone private copies does not work if the data is ever-changing. Peripherals can also be shared. Some peripherals are expensive, remote, and/or infrequently used so it is not justifiable to give each user a dedicated device. These devices may include include file servers, large format printers, and drum scanners. Computers themselves can be shared and workload can be distributed amongst idle machines. Finally, networked machines are useful for supporting person-to-person networking: exchanging messages, files, and other information.

Design challenges

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.

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.

A centralized model 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.

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 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. 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 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.

Thick and thin 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. With thin clients, there is no need for on-device administration. 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. 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 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. 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.

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 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 scaling computing power itself. In the most intelligent case, an operating system would be able to automatically start processes on idle machines and even migrate processes to systems 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. In the processor pool model, we maintain a collection of systems that can be dynamically assigned to processes on demand.

References (partial)

- Andrew S. Tanenbaum, Maarten Van Steen, *Distributed Systems: Principles and Paradigms* (2nd Edition). © 2006 Prentice Hall
- B. Clifford Neuman. *Scale in Distributed Systems*. In Readings in Distributed Computing Systems. IEEE Computer Society Press
- George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair, *Distributed Systems: Concepts and Design* (5th edition). © 2011 Addison Wesley
- Intel Streaming SIMD Extensions Technology, Intel, September 10, 2018.

This is an update of an original document written on September 14, 2012.

* * *

1. 2.94 Mbps is the speed you get when using the Xerox Alto computer's 170 nanosecond clock, ticking twice per bit. ↩
2. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html ↩