COMMUNICATION

9(21), January 1, 2014

ISSN 2278–5469
EISSN 2278–5450

Discovery

Distributed operating system: a perspective

Annu, Himanshu rehani

Department of Computer Science and Engineering, Dronacharya College of Engineering, Khentawas, Farukhnagar, Gurgaon, India

Publication History
Received: 12 November 2013
Accepted: 19 December 2013
Published: 1 January 2014

Citation
Annu, Himanshu Rehani. Distributed operating system: a perspective. *Discovery*, 2014, 9(21), 21-33

ABSTACT

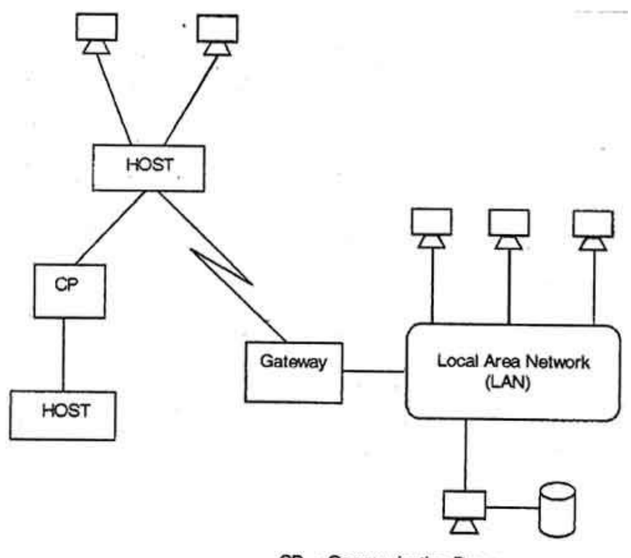
A distributed operating system is software over a collection of independent, networked, communicating, and physically separate computational nodes. Individual nodes each hold a specific software subset of the global aggregate operating system. Each subset is a composite of two distinct service provisioners. First is a ubiquitous minimal kernel, or microkernel, that directly controls that node's hardware. Second is a higher-level collection of system management components that coordinate the node's individual and collaborative activities. These components abstract microkernel functions and support user applications. A collection of independent computers which can cooperate, but which appear to users of the system as a uniprocessor computer the microkernel and the management components collection work together. They support the system's goal of integrating multiple resources and processing functionality into an efficient and stable system. This seamless integration of individual nodes into a global system is referred to as transparency, or single system image; describing the illusion provided to users of the global system's appearance as a single computational entity. These systems are referred as loosely coupled systems where each processor has its own local memory and processors communicate with one another through various communication lines, such as high speed buses or telephone lines. By loosely coupled systems, we mean that such computers possess no hardware connections at the CPU - memory bus level, but are connected by external interfaces that run under the control of software. The Distributed Os involves a collection of autonomous computer systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed Os provides a virtual machine abstraction to its users and wide sharing of resources like as computational capacity, I/O and files etc. The users of a true distributed system should not know, on which machine their programs are running and where their files are stored.

1. INTRODUCTION

A distributed operating system is an operating system that runs on several machines whose purpose is to provide a useful set of services, generally to make the collection of machines behave more like a single machine. The distributed operating system plays the same role in making the collective resources of the machines more usable that a typical single-machine operating system plays in making that machine's resources more usable. Usually, the machines controlled by a distributed operating system are connected by a relatively high quality network, such as a high speed local area network. Most commonly, the participating nodes of the system are in a relatively small geographical area, something between an office and a campus. Distributed operating systems typically run cooperatively on all machines whose resources they control. These machines might be capable of independent operation, or they might be usable merely as resources in the distributed system. In some architecture, each machine is an equally powerful peer as all the others. In other architectures, some machines are permanently designated as master or are given control of particular resources. In yet others, elections or other selection mechanisms are used to designate some machines as having special roles, often controlling roles. A parallel operating system is usually defined as running on specially designed parallel processing hardware. It usually works on the assumption that elements of the hardware (such as the memory) are tightly coupled. Often, the machine is expected to be devoted to running a single task at very high speed. A distributed operating system is usually defined as running on more loosely coupled hardware. Unlike parallel operating systems, distributed operating systems are intended to make a collection of resources on multiple machines usable by a set of loosely cooperating users running independent tasks. Network operating systems are sometimes regarded as systems that attempt merely to make the network connecting the machines more usable, without regard for some of the larger problems of building effective distributed systems. Although many interesting research distributed operating systems have been built since the 1970s, and some systems have been in use for many years, they have not displaced traditional operating systems designed primarily to support single machines; however, some of the components originally built for distributed operating systems have become commonplace in today's systems, notably services to access files stored on remote machines. The failure of distributed operating systems to capture a large share of the marketplace may be primarily due to our lack of understanding on how to build them, or perhaps their lack of popularity stems from users not really needing many distributed services not already provided.

1.1. Distributed operating systems

Distributed operating systems are also an important field for study because they have helped drive general research in distributed systems. Replicated data systems, authentication services such as Kerberos, agreement protocols, methods of providing causal ordering in communications, voting and consensus protocols, and many other distributed services have been developed to support distributed operating systems, and have found varying degrees of success outside of that field. Popular distributed component services like CORBA owe some of their success to applying hard lessons learned by researchers in distributed operating systems. Increasingly, cooperative applications and services run across the Internet, and they face similar problems to those seen and frequently solved in the realm of distributed operating systems. Distributed operating systems are hard to design because they face inherently hard problems, such as distributed consensus and synchronization. Further, they must properly trade off issues of performance, user interfaces, reliability, and simplicity. The relative scarcity of such systems, and the fact that most commercial operating systems' design still focuses on single-machine systems, suggests that no distributed operating system yet developed has found the proper trade-off among these issues. Research continues in distributed operating systems, particularly in certain critical elements of them that have obvious value, especially file systems and other forms of data sharing. Other continuing research in distributed operating systems focuses on their use in important special cases, such as high-performance clustered servers and grid computing. Cloud computing is a recent development closely related to distribute operating systems. The increasing popularity of smart phones and tablets points out further need, if not for distributed operating systems, than at least for better methods to allow mobile devices to share their resources and work cooperatively. The emerging field of ubiquitous computing offers different hardware, networking, and application characteristics likely to spur further research on distributed operating systems. Peer systems, currently used primarily to share data, are also likely to spur further research in distributed operating systems issues. Sensor networks are another form of highly specialized distributed system that has benefited from the lessons of distributed operating systems.



CP → Communication Processors

1.2. Examples of Distributed Operating Systems

LOCUS and MICROS are the best examples of distributed operating systems. Using LOCUS operating system it was possible to access local and distant files in uniform manner. This feature enabled a user to log on any node of the network and to utilize the resources in a network without the reference of his/her location. MICROS provided sharing of resources in an automatic manner. The jobs were assigned to different nodes of the whole system to balance the load on different nodes.

1. IRIX operating system; is the implementation of UNIX System V, Release 3 for Silicon Graphics multiprocessor workstations.
2. DYNIX operating system running on Sequent Symmetry multiprocessor computers.
3. AIX operating system for IBM RS/6000 computers.
4. Solaris operating system for SUN multiprocessor workstations.
5. Mach/OS is a multithreading and multitasking UNIX compatible operating system;
6. OSF/1 operating system developed by Open Foundation Software: UNIX compatible.

2. DISTRIBUTED COMPUTING MODELS

2.1. Three basic distributions

To better illustrate this point, examine three system architectures; centralized, decentralized, and distributed. In this examination, consider three structural aspects: organization, connection, and control. Organization describes a system's physical arrangement characteristics. Connection covers the communication pathways among nodes. Control manages the operation of the earlier two considerations.

2.1.1. Organization

A centralized system has one level of structure, where all constituent elements directly depend upon a single control element. A decentralized system is hierarchical. The bottom level unites subsets of a system’s entities. These entity subsets in turn combine at higher levels, ultimately culminating at a central master element. A distributed system is a collection of autonomous elements with no concept of levels.

2.1.2. Connection

Centralized systems connect constituents directly to a central master entity in a hub and spoke fashion. A decentralized system (aka network system) incorporates direct and indirect paths between constituent elements and the central entity. Typically this is configured as a hierarchy with only one shortest path between any two elements. Finally, the distributed operating system requires no pattern; direct and indirect connections are possible between any two elements. Consider the 1970s phenomena of ”string art” or a paragraph drawing as a fully connected system, and the spider’s web or the Interstate Highway System between U.S. cities as examples of a partially connected system.

2.1.2. Control

Centralized and decentralized systems have directed flows of connection to and from the central entity, while distributed systems communicate along arbitrary paths. This is the pivotal notion of the third consideration. Control involves allocating tasks and data to system elements balancing efficiency, responsiveness and complexity. Centralized and decentralized systems offer more control, potentially easing administration by limiting options. Distributed systems are more difficult to explicitly control, but scale better horizontally and offer fewer points of system-wide failure. The associations conform to the needs imposed by its design but not by organizational limitations.

3. DESIGN ISSUES

3.1. Transparency

Transparency or single-system image refers to the ability of an application to treat the system on which it operates without regard to whether it is distributed and without regard to hardware or other implementation details. Many areas of a system can benefit from transparency, including access, location, performance, naming, and migration. The consideration of transparency directly effects decision making in every aspect of design of a distributed operating system. Transparency can impose certain requirements and/or restrictions on other design considerations. Systems can optionally violate transparency to varying degrees to meet specific application requirements. For example, a distributed operating system may present a hard drive on one computer as "C:" and a drive on another computer as "G:". The user does not require any knowledge of device drivers or the drive's location; both devices work the same way, from the application's perspective. A less transparent interface might require the application to know which computer hosts the drive.

3.1.1. Transparency domains

3.1.1.1. Location transparency

Location transparency comprises two distinct aspects of transparency, naming transparency and user mobility. Naming transparency requires that nothing in the physical or logical references to any system entity should expose any indication of the entity's location, or its local or remote relationship to the user or application. User mobility requires the consistent referencing of system entities, regardless of the system location from which the reference originates.

3.1.1.2. Access transparency

Local and remote system entities must remain indistinguishable when viewed through the user interface. The distributed operating system maintains this perception through the exposure of a single access mechanism for a system entity, regardless of that entity being local or remote to the user. Transparency dictates that any differences in methods of accessing any particular system entity—either local or remote—must be both invisible to, and undetectable by the user.

3.1.1.3. Migration transparency

Resources and activities migrate from one element to another controlled solely by the system and without user/application knowledge or action.

3.1.1.4 Replication transparency

The process or fact that a resource has been duplicated on another element occurs under system control and without user/application knowledge or intervention.

3.1.1.5. Concurrency transparency

Users/applications are unaware of and unaffected by the presence/activities of other users.

3.1.1.6. Failure transparency

The system is responsible for detection and remediation of system failures. No user knowledge/action is involved other than waiting for the system to resolve the problem.

3.1.1.7. Performance Transparency

The system is responsible for the detection and remediation of local or global performance shortfalls. Note that system policies may prefer some users/user classes/tasks over others. No user knowledge or interaction is involved.

3.1.1.8. Size/Scale transparency

The system is responsible for managing its geographic reach, number of nodes, and level of node capability without any required user knowledge or interaction.

3.1.1.9. Revision transparency

The system is responsible for upgrades and revisions and changes to system infrastructure without user knowledge or action.

3.1.2. Control transparency

The system is responsible for providing all system information, constants, properties, configuration settings, etc. in a consistent appearance, connotation, and denotation to all users and applications.

3.1.2.1 Data transparency

The system is responsible for providing data to applications without user knowledge or action relating to where the system stores it.

3.1.2.2. Parallelism transparency

The system is responsible for exploiting any ability to parallelize task execution without user knowledge or interaction. Arguably the most difficult aspect of transparency, and described by Tanenbaum as the "Holy grail" for distributed system designers.

3.2. Reliability

Distributed OS can provide the necessary resources and services to achieve high levels of reliability, or the ability to prevent and/or recover from errors. Faults are physical or logical defects that can cause errors in the system. For a system to be reliable, it must somehow overcome the adverse effects of faults. The primary methods for dealing with faults include fault avoidance, fault tolerance, and fault detection and recovery. Fault avoidance covers proactive measures taken to minimize the occurrence of faults. These proactive measures can be in the form of transactions, replication and backups. Fault tolerance is the ability of a system to continue operation in the presence of a fault. In the event, the system should detect and recover full functionality. In any event, any actions taken should make every effort to preserve the single system image. One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The idea is that if a machine goes down, some other machine takes over the job. In other words, theoretically the overall system reliability could be the Boolean OR of the component reliabilities. For example, with four file servers, each with a 0.95 chance of being up at any instant, the probability of all four being down simultaneously is $0.05^4 = 0.000006$, so the probability of at least one being available is 0.999994, far better than that of any individual server. It is important to distinguish various aspects of reliability.

3.2.1. Availability

Availability as we have just seen refers to the fraction of time that the system is usable. Lamppost’s system apparently did not score well in that regard. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another tool for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the slack. A highly reliable system must be highly available, but that is not enough. Data entrusted to the system must not be lost or garbled in any way, and if files are stored redundantly on multiple servers, all the copies must be kept consistent. In general, the more copies that are kept, the better the availability, but the greater the chance that they will be inconsistent, especially if updates are frequent. Another aspect of overall reliability is security. Files and other resources must be protected from unauthorized usage. Although the same issue occurs in single-processor systems, in distributed systems it is more severe. In a single-processor system, the user logs in and is authenticated. From then on, the system knows who the user is and can check whether each attempted access is legal. In a distributed system, when a message comes in to a server asking for something, the server has no simple way of determining who it is from. No name or identification field in the message can be trusted, since the sender may be lying another issue relating to reliability is fault tolerance. Suppose that a server crashes and then quickly reboots. what happens? Does the server crash bring users down with it? If the server has tables containing important information about ongoing activities, recovery will be difficult at best. In general, distributed systems can be designed to mask failures, that is, to hide them from the users. If a file service or other service is actually constructed from a group of closely cooperating servers, it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. Of course, the trick is to arrange this cooperation so that it does not add substantial overhead to the system in the normal case, when everything is functioning correctly. Availability is the fraction of time during which the system can respond to requests.

3.2.2. Performance

Many benchmark metrics quantify performance; throughput, response time, job completions per unit time, system utilization, etc. With respect to a distributed OS, performance most often distills to a balance between process parallelism and IPC.] Managing the task granularity of parallelism in a sensible relation to the messages required for support is extremely effective. Also, identifying when it is more beneficial to migrate a process to its data, rather than copy the data, is effective as well.

3.2.3. Synchronization

Cooperating concurrent processes have an inherent need for synchronization, which ensures that changes happen in a correct and predictable fashion. Three basic situations that define the scope of this need: one or more processes must synchronize at a given point for one or more other processes to continue, one or more processes must wait for an asynchronous condition in order to continue, or a process must establish exclusive access to a shared resource. Improper synchronization can lead to multiple failure modes including loss of atomicity, consistency, isolation and durability, deadlock, live lock and loss of serializability.

3.3. Flexibility

The second key design issue is flexibility. It is important that the system be flexible because we are just beginning to learn about how to build distributed systems. It is likely that this process will incur many false starts and considerable backtracking. Design decisions that now seem reasonable may later prove to be wrong. It is hard to imagine anyone arguing in favor of an inflexible system. However, things are not as simple as they seem. There are two schools of thought concerning the structure of distributed systems. One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from user-level servers. These two models, known as the monolithic kernel and microkernel, respectively.

The monolithic kernel is basically today's centralized operating system augmented with networking facilities and the integration of remote services. Most system calls are made by trapping to the kernel, having the work performed there, and having the kernel return the desired result to the user process. With this approach, most machines have disks and manage their own local file systems. Many distributed systems that are extensions or imitations of UNIX use this approach because UNIX itself has a large, monolithic kernel. If the monolithic kernel is the reigning champion, the microkernel is the up-and-coming challenger. Most distributed systems that have been designed from scratch use this method. The microkernel is more flexible because it does almost nothing. It basically provides just four minimal services:

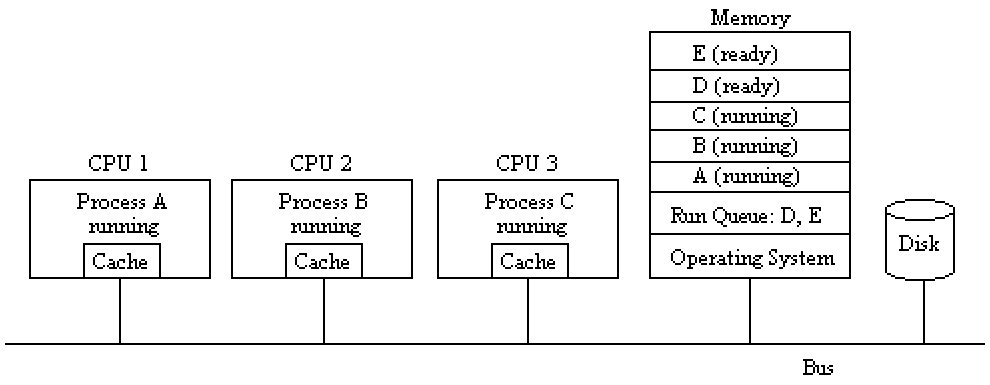
1. An interposes communication mechanism.
2. Some memory management.
3. A small amount of low-level process management and scheduling.
4. Low-level input/output.

In particular, unlike the monolithic kernel, it does not provide the file system, directory system, full process management, or much system call handling. The services that the microkernel does provide are included because they are difficult or expensive to provide anywhere else. The goal is to keep it small. All the other operating system services are generally implemented as user-level servers. To look up a name, read a file, or obtain some other service, the user sends a message to the appropriate server, which then does the work and returns the result. The advantage of this method is that it is highly modular: there is a well-defined interface to each service (the set of messages the server understands), and every service is equally accessible to every client, independent of location. In addition, it is easy to implement, install, and debug new services, since adding or changing a service does not require stopping the system and booting a new kernel, as is the case with a monolithic kernel. It is precisely this ability to add, delete, and modify services that gives the microkernel its flexibility. Furthermore, users who are not satisfied with any of the official services are free to write their own. As a simple example of this power, it is possible to have a distributed system with multiple file servers, one supporting MS-DOS file service and another supporting UNIX file service. Individual programs can use either or both, if they choose. In contrast, with a monolithic kernel, the file system is built into the kernel, and users have no choice but to use it. The only potential advantage of the monolithic kernel is performance. Trapping to the kernel and doing everything there may well be faster than sending messages to remote servers. However, a detailed comparison of two distributed operating systems, one with a monolithic kernel (Sprite), and one with a microkernel (Amoeba), has shown that in practice this advantage is nonexistent .Other factors tend to dominate, and the small amount of time required to send a message and get a reply (typically, about 1 m sec) is usually negligible.

3.4 Scalability

Most current distributed systems are designed to work with a few hundred CPUs. It is possible that future systems will be orders of magnitude larger, and solutions that work well for 200 machines will fail miserably for 200,000,000. Consider the following. The French PTT (Post, Telephone and Telegraph administration) is in the process of installing a terminal in every household and business in France. The terminal, known as a minutes, will allow online access to a data base containing all the telephone numbers in France, thus eliminating the need for printing and distributing expensive telephone books. It will also vastly reduce the need for information operators who do nothing but give out telephone numbers all day. It has been calculated that the system will pay for itself within a few years. If the system works in France, other countries will inevitably adopt similar systems. Once all the terminals are in place, the possibility of also using them for electronic mail (especially in conjunction with printers) is clearly present. Since postal services lose a huge amount of money in every country in the world, and telephone services are enormously profitable, there are great incentives to having electronic mail replace paper mail. Next comes interactive access to all kinds of data bases and services, from electronic banking to reserving places in planes, trains, hotels, theaters, and restaurants, to name just a few. Before long, we have a distributed system with tens of millions of users. Although little is known about such huge distributed systems, one guiding principle is clear:

avoid centralized components, tables, and algorithms. Having a single mail server for 50 million users would not be a good idea. Even if it had enough CPU and storage capacity, the network capacity into and out of it would surely be a problem. Furthermore, the system would not tolerate faults well. A single power outage could bring the entire system down. Finally, most mail is local. Having a message sent by a user in Marseille to another user two blocks away pass through a machine in Paris is not the way to go.



Concept	Example
Centralized components	A single mail server for all users
Centralized tables	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Centralized tables are almost as bad as centralized components. How should one keep track of the telephone numbers and addresses of 50 million people? Suppose that each data record could be fit into 50 characters. A single 2.5-gigabyte disk would provide enough storage. But here again, having a single data base would undoubtedly saturate all the communication lines into and out of it. It would also be vulnerable to failures (a single speck of dust could cause a head crash and bring down the entire directory service). Furthermore, here too, valuable network capacity would be wasted shipping queries far away for processing. Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing. The trouble is that collecting and transporting all the input and output information would again be a bad idea for the reasons discussed above. In fact, any algorithm that operates by collecting information from all sites sends it to a single machine for processing, and then distributes the results must be avoided. Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

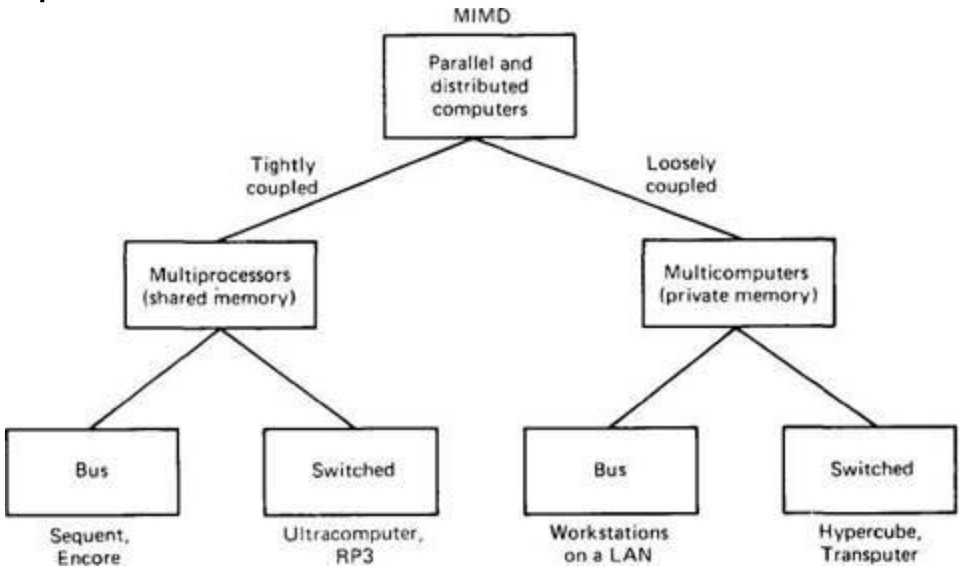
1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious, but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally is tricky.

4. TRUE DISTRIBUTED SYSTEMS

Network operating systems are loosely-coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that such a system consists of numerous computers. Each can run its own operating system and do whatever its owner wants. There is essentially no coordination at all, except for the rule that client-server traffic must obey the system's protocols. The next evolutionary step beyond this is tightly-coupled software on the same loosely-coupled (i.e., multicomputer) hardware. The goal of such a system is to create the illusion in the minds of the users that the entire network of computers is a single timesharing system, rather than a collection of distinct machines. Some authors refer to this property as the single-system image. Others put it slightly differently, saying that a distributed system is one that runs on a collection of networked machines but acts like a virtual uniprocessor. No matter how it is expressed, the essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. No current system fulfills this requirement entirely, but a number of candidates are on the horizon. These will be discussed later in the book. What are some characteristics of a distributed system? To start with, there must be a single, global interposes communication mechanism so that any process can talk to any other process. It will not do to have different mechanisms on different machines or different mechanisms for local communication and remote communication. There must also be a global protection scheme. Mixing access control lists, the UNIX® protection bits, and capabilities will not give a single system image. Process management must also be the same everywhere. How processes are created, destroyed, started, and stopped must not vary from machine to machine. In short, the idea behind network operating systems, namely that any machine can do whatever it wants to as long as it obeys the standard protocols when engaging in client-server communication, is not enough. Not only must there be a single set of system calls available on all machines, but these calls must be designed so that they make sense in a distributed environment. The file system must look the same everywhere, too. Having file names restricted to 11 characters in some locations and being unrestricted in others is undesirable. Also, every file should be visible at every location, subject to protection and security constraints, of course. As a logical consequence of having the same system call interface everywhere, it is normal that identical kernels run on all the CPUs in the system. Doing so makes it easier to coordinate activities that must be global. For example, when a process has to be started up, all the kernels have to cooperate in finding the best place to execute it. In addition, a global file system is needed. Nevertheless, each kernel can have considerable control over its own local resources. For example, since there is no shared memory, it is logical to allow each kernel to manage its own memory. For example, if swapping or paging is used, the kernel on each CPU is the logical place to determine what to swap or page. There is no reason to centralize this authority. Similarly, if multiple processes are running on some CPU, it makes sense to do the scheduling right there, too.

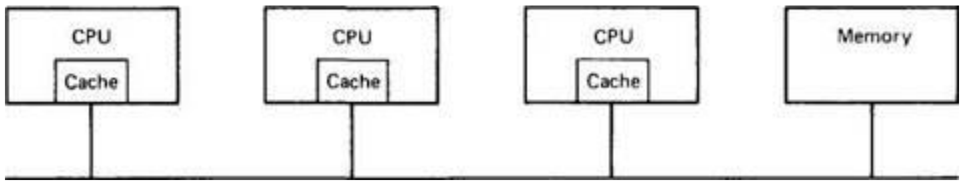
4.1. Hardware Concepts



Even though all distributed systems consist of multiple CPUs, there are several different ways the hardware can be organized, especially in terms of how they are interconnected and how they communicate. In this section we will take a brief look at distributed system hardware, in particular, how the machines are connected together. In the next section we will examine some of the software issues related to distributed systems. Various classification schemes for multiple CPU computer systems have been proposed over the years, but none of them have really caught on and been widely adopted. Probably the most frequently cited taxonomy is Flynn's

(1972), although it is fairly rudimentary. Flynn picked two characteristics that he considered essential: the number of instruction streams and the number of data streams. A computer with a single instruction stream and a single data stream is called SISD. All traditional uniprocessor computers (i.e., those having only one CPU) fall in this category, from personal computers to large mainframes. The next category is SIMD, single instruction stream, multiple data stream. This type refers to array processors with one instruction unit that fetches an instruction, and then commands many data units to carry it out in parallel, each with its own data. These machines are useful for computations that repeat the same calculation on many sets of data, for example, adding up all the elements of 64 independent vectors. Some supercomputers are SIMD. The next category is MISD, multiple instruction stream, single data stream. No known computers fit this model. Finally, comes MIMD, which essentially means a group of independent computers, each with its own program counter, program, and data. All distributed systems are MIMD, so this classification system is not tremendously useful for our purposes. Although Flynn stopped here, we will go further. In Figure 1-4, we divide all MIMD computers into two groups: those that have shared memory, usually called multiprocessors, and those that do not, sometimes called multicomputer. The essential difference is this: in a multiprocessor, there is a single virtual address space that is shared by all CPUs. If any CPU writes, for example, the value 44 to address 1000, any other CPU subsequently reading from its address 1000 will get the value 44. All the machines share the same memory.

In contrast, in a multicomputer, every machine has its own private memory. If one CPU writes the value 44 to address 1000, when another CPU reads address 1000 it will get whatever value was there before. The write of 44 does not affect its memory at all. A common example of a multicomputer is a collection of personal computers connected by a network. Each of these categories can be further divided based on the architecture of the interconnection network. In Fig. 1-4 we describe these two categories as bus and switched. By bus we mean that there is a single network, backplane, bus, cable, or other medium that connects all the machines. Cable television uses a scheme like this: the cable company runs a wire down the street, and all the subscribers have taps running to it from their television sets. Switched systems do not have a single backbone like cable television. Instead, there are individual wires from machine to machine, with many different wiring patterns in use. Messages move along the wires, with an explicit switching decision made at each step to route the message along one of the outgoing wires. The worldwide public telephone system is organized in this way. Another dimension to our taxonomy is that in some systems the machines are tightly coupled and in others they are loosely coupled. In a tightly-coupled system, the delay experienced when a message is sent from one computer to another is short, and the data rate is high; that is, the number of bits per second that can be transferred is large. In a loosely-coupled system, the opposite is true: the intermachine message delay is large and the data rate is low. For example, two CPU chips on the same printed circuit board and connected by wires etched onto the board are likely to be tightly coupled, whereas two computers connected by a 2400 bit/sec modem over the telephone system are certain to be loosely coupled. Tightly-coupled systems tend to be used more as parallel systems (working on a single problem) and loosely-coupled ones tend to be used as distributed systems (working on many unrelated problems), although this is not always true. One famous counterexample is a project in which hundreds of computers all over the world worked together trying to factor a huge number (about 100 digits). Each computer was assigned a different range of divisors to try, and they all worked on the problem in their spare time, reporting the results back by electronic mail when they finished. On the whole, multiprocessors tend to be more tightly coupled than multi-computers, because they can exchange data at memory speeds, but some fibrotic based multicomputer can also work at memory speeds. Despite the vagueness of the terms "tightly coupled" and "loosely coupled," they are useful concepts; just as saying "Jack is fat and Jill is thin" conveys information about girth even though one can get into a fair amount of discussion about the concepts of "fatness" and "thinness." In the following four sections, we will look at the four categories of Fig. 1-4 in more detail, namely bus multiprocessors, switched multiprocessors, bus multicomputer, and switched multicomputer. Although these topics are not directly related to our main concern, distributed operating systems, they will shed some light on the subject because as we shall see, different categories of machines use different kinds of operating systems.



4.2. Bus-Based Multiprocessors

Bus-based multiprocessors consist of some number of CPUs all connected to a common bus, along with a memory module. A simple configuration is to have a high-speed backplane or motherboard into which CPU and memory cards can be inserted. A typical bus has 32 or 64 address lines, 32 or 64 data lines, and perhaps 32 or more control lines, all of which operate in parallel. To read a word of memory, a CPU puts the address of the word it wants on the bus address lines, then puts a signal on the appropriate control lines to indicate that it wants to read. The memory responds by putting the value of the word on the data lines to allow the requesting CPU to read it in. Writes work in a similar way. Since there is only one memory, if CPU A writes a word to memory and then CPU B reads that word back a microsecond later, B will get the value just written. A memory that has this property is said to be coherent. Coherence plays an important role in distributed operating systems in a variety of ways that we will study later. The problem with this scheme is that with as few as 4 or 5 CPUs, the bus will usually be overloaded and performance will drop drastically. The solution is to add a high-speed cache memory between the CPU and the bus. The cache holds the most recently accessed words. All memory requests go through the cache. If the word requested is in the cache, the cache itself responds to the CPU, and no bus request is made. If the cache is large enough, the probability of success, called the hit rate, will be high, and the amount of bus traffic per CPU will drop dramatically, allowing many more CPUs in the system. Cache sizes of 64K to 1M are common, which often gives a hit rate of 90 percent or more.

However, the introduction of caches also brings a serious problem with it. Suppose that two CPUs, A and B, each read the same word into their respective caches. Then A overwrites the word. When B next reads that word, it gets the old value from its cache, not the value A just wrote. The memory is now incoherent, and the system is difficult to program. Many researchers have studied this problem, and various solutions are known. Below we will sketch one of them. Suppose that the cache memories are designed so that whenever a word is written to the cache, it is written through to memory as well. Such a cache is, not surprisingly, called a write-through cache. In this design, cache hits for reads do not cause bus traffic, but cache misses for reads, and all writes, hits and misses, cause bus traffic. In addition, all caches constantly monitor the bus. Whenever a cache sees a write occurring to a memory address present in its cache, it either removes that entry from its cache, or updates the cache entry with the new value. Such a cache is called a snoopy cache (or sometimes, a snooping cache) because it is always "snooping" (eavesdropping) on the bus. A design consisting of snoopy write-through caches is coherent and is invisible to the programmer. Nearly all bus-based multiprocessors use either this architecture or one closely related to it. Using it, it is possible to put about 32 or possibly 64 CPUs on a single bus.

5. DISTRIBUTED SYSTEMS ADVANTAGES

Sharing of Data and Resources, Reliability, Communication, Computation speedup, Flexibility Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with. A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using shared memory architecture), multicomputer connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems. A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own memory, hard disk. There are some shared resources such files and printers. If the interconnection network broke down, individual computers could be used but without some features like printing to a non-local printer.

5.1. Disadvantages of Distributed Systems

Although distributed systems have their strengths, they also have their weaknesses. We have already hinted at the worst problem: software. With the current state-of-the-art, we do not have much experience in designing, implementing, and using distributed software. What kinds of operating systems, programming languages, and applications are appropriate for these systems? How much should the users know about the distribution? How much should the system do and how much should the users do? The experts differ (not that this is unusual with experts, but when it comes to distributed systems, they are barely on speaking terms). As more research is done, this problem will diminish, but for the moment it should not be underestimated. A second potential problem is due to the communication network. It can lose messages, which requires special software to be able to recover, and it can become overloaded. When the network saturates, it must either be replaced or a second one must be added. In both cases, some portion of one or more buildings may have to be rewired at great expense, or network interface boards may have to be replaced (e.g., by fiber optics). Once the system comes to depend on the network, its loss or saturation can negate most of the advantages the distributed system was built to achieve. Finally, the easy sharing of data, which we described above as an advantage, may turn out to be a two-edged sword. If people can conveniently access data all over the system, they may equally be able to conveniently access data that they have no business looking at. In other words, security is often a problem. For data that must be kept secret at all costs, it is often preferable to have a dedicated, isolated personal computer that has no network connections to any other machines, and is kept in a locked room with a secure safe in which all the floppy disks are stored. Despite these potential problems, many people feel that the advantages outweigh the disadvantages, and it is expected that distributed systems will become increasingly important in the coming years. In fact, it is likely that within a few years, most organizations will connect most of their computers into large distributed systems to provide better, cheaper, and more convenient service for the users. An isolated computer in a medium-sized or large business or other organization will probably not even exist in ten years.

Item	Description
Software	Little software exists at present for distributed systems
Networking	The network can saturate or cause other problems
Security	Easy access also applies to secret data

6. CHALLENGES IN BUILDING DISTRIBUTED OPERATING SYSTEMS

One core problem for distributed operating system designers is concurrency and synchronization. These issues arise in single-machine operating systems, but they are easier to solve there. Typical single-machine systems run a single thread of control simultaneously, simplifying many synchronization problems. The advent of multicore machines is complicating this issue, but most multicore machines have relatively few cores, lessoning the problem. Further, they typically have shared access to memory, registers, or other useful physical resources that are directly accessible by all processes that they must synchronize. These shared resources allow use of simple and fast synchronization primitives, such as semaphores. Even modern machines that have multiple processors typically include hardware that makes it easier to synchronize their operations. Distributed operating systems lack these advantages. Typically, they must control a collection of processors connected by a network, most often a local area network (LAN), but occasionally a network with even more difficult characteristics. The access time across this network is orders of magnitude larger than the access time required to reach local main memory and even more orders of magnitude larger than that required to reach information in a local processor cache or register. Further, such networks are not as reliable as a typical bus, so messages are more likely to be lost or corrupted. At best, this unreliability increases the average access time. This imbalance means that running blocking primitives across the network is often infeasible. The performance implications for the individual component systems and the system as a whole do not permit widespread use of such primitives. Designers must choose between looser synchronization (leading to odd user-visible behaviors and possibly fatal system inconsistencies) and sluggish performance. The increasing gap between processor and network speeds suggests that this effect will only get worse. Theoretical results in distributed systems are discouraging. Research on various forms of the Byzantine General problem and other formulations of the problems of reaching decisions in distributed systems has provided surprising results with bad implications for the possibility of providing perfect synchronization of such systems. Briefly, these results suggest that reaching a distributed decision is not always possible in common circumstances. Even when it is possible, doing so in unfavorable conditions is very expensive and tricky. Although most distributed systems can be designed to operate in more favorable circumstances than these gloomy theoretical results describe (typically by assuming less drastic failure modes or less absolute need for complete consistency), experience has shown that even pragmatic algorithm design for this environment is difficult. A further core problem is providing transparency. Transparency has various definitions and aspects, but at a high level it simply refers to the degree to which the operating system disguises the distributed nature of the system. Providing a high degree of transparency is good because it shields the user from the complexities of distribution. On the other hand, it sometimes hides more than it should, it can be expensive and tricky to provide, and ultimately it is not always possible. A key decision in designing a distributed operating system is how much transparency to provide, and where and when to provide it. A related problem is that the hardware, which the distributed operating system must virtualized, is more varied. A distributed operating system must not only make a file on disk appear to be in the main memory, as a typical operating system does, but must make a file on a different machine appear to be on the local machine, even if it is simultaneously being accessed on yet a third machine. The system should not just make a multi-machine computation appear to run on a single machine, but should provide observers on all machines with the illusion that it is running only on their machine. Distributed operating systems also face challenging problems because they are typically intended to continue correct operation despite failure of some of their components. Most single-machine operating systems provide very limited abilities to continue operation if key components fail. They are certainly not expected to provide useful service if their processor crashes. A single processor crash in a distributed operating system should allow the remainder of the system to continue operations largely unharmed. Achieving this ideal can be extremely challenging. If the topology of the network connecting the system's component nodes allows the network to split into disjoint pieces, the system might also need to continue operation in a partitioned mode and would be expected to rapidly reintegrate when the partitions merge. The security problems of a distributed operating system are also harder. First, data typically moves over a network, sometimes over a network that the distributed operating system itself does not directly control. This network may be subject to eavesdropping or malicious insertion and alteration of messages. Even if protected by cryptography, denial of service attacks may cause disconnections or loss of critical messages. Second, access control and resource management mechanisms on single machines typically take advantage of hardware that helps keep processes separate, such as page tables. Distributed operating systems cannot rely on this advantage. Third, distributed operating systems are typically expected to provide some degree of local control to users on their individual machines, while still enforcing general access control mechanisms. When an individual user is legitimately able to access any bytes stored anywhere on his own machine, preventing him from accessing data that belongs to others is a much harder problem, particularly if the system strives to provide controlled high-performance access to that data. Distributed operating systems must often address the issue of local autonomy. In many (but not all) architectures, the distributed system is composed of workstations whose primary job is to support one particular user. The distributed system must balance the needs of the entire collection of supported users against the natural expectation that one's machine should be under one's own control. The local autonomy question has clear security implications, but also relates to how resources are allocated, how scheduling is done, and other issues.

In many cases, distributed operating systems are expected to run on heterogeneous hardware. Although commercial convergence on a small set of popular processors has reduced this problem to some extent, the wide variety of peripheral devices and customizations of system settings provided by today's operating systems often makes supposedly identical hardware behave radically differently. If a distributed operating system cannot determine whether running the same operation on two different component nodes produces the same result, it will face difficulties in providing transparency and consistency. All the previously mentioned problems are exacerbated if the system scale becomes sufficiently large. Many useful distributed algorithms scale poorly, because the number of messages they require faces combinatorial explosion, or because the delays required to include large numbers of nodes in computations become unreasonable, or because data structures grow in proportion to the number of participants. High scale ensures that partial failures will become more common, and that low probability events will begin to pop up every so often. High scale might also imply that the distributed operating system must operate away from the relatively friendly world of the LAN, leading to greater heterogeneity and uncertainty in communications.

An entirely different paradigm of building system software for distributed systems can avoid some of these difficulties. Sensor networks, rather than performing general purpose computing, are designed only to gather information from sensors and send it to places that need it. The nodes in a sensor network are typically very simple and have low power in many dimensions, from CPU speed to battery. As a result, while inherently distributed systems, sensor network nodes must run relatively simple code. Operating systems designed for sensor networks, like Tiny OS, are thus themselves extremely simple. By proper design of the operating system and algorithms that perform the limited applications, a sensor network achieves a cooperative distributed goal without worrying about many of the classic issues of distributed operating systems, such as tight synchronization, data consistency, and partial failure. This approach does not seem to offer an alternative when one is designing a distributed operating system for typical desktop or server machines, but may prove to be a powerful tool for other circumstances in which the nodes in the distributed systems need only do very particular and limited tasks. Other limited versions of distributed operating system also avoid many of the worst difficulties faced in the general case. In cloud computing, for example, the provider of the cloud does not himself have to worry about maintaining transparency or consistency among the vast number of nodes he supports. His distributed systems problems are more limited, relating to management of large numbers of nodes, providing strong security between the users of portions of his system, ensuring fair and fast use of the network, and, at most, providing some basic distributed system primitives to his users. By expecting the users or middleware to customize the basic node operations he provides to suit their individual distributed system needs, the cloud provider offloads many of the most troublesome problems in distributed operating systems. Since the majority of users of cloud computing don't need those problems solved, anyway, this approach suits both the requirements of the cloud provider.

7. CONCLUSION

Distributed systems consist of autonomous CPUs that work together to make the complete system look like a single computer. They have a number of potential selling points, including good price/performance ratios, the ability to match distributed applications well, potentially high reliability, and incremental growth as the workload grows. They also have some disadvantages, such as more complex software, potential communication bottlenecks, and weak security. Nevertheless, there is considerable interest worldwide in building and installing them. Modern computer systems often have multiple CPUs. These can be organized as multiprocessors (with shared memory) or as multicomputer (without shared memory). Both types can be bus-based or switched. The former tend to be tightly coupled, while the latter tend to be loosely coupled. The software for multiple CPU systems can be divided into three rough classes. Network operating systems allow users at independent workstations to communicate via a shared file system but otherwise leave each user as the master of his own workstation. Distributed operating systems turn the entire collection of hardware and software into a single integrated system, much like a traditional timesharing system. Shared-memory multiprocessors also offer a single system image, but do so by centralizing everything, so there really is only a single system. Shared-memory multiprocessors are not distributed systems. Distributed systems have to be designed carefully, since there are many pitfalls for the unwary. A key issue is transparency — hiding all the distribution from the users and even from the application programs. Another issue is flexibility. Since the field is only now in its infancy, the design should be made with the idea of making future changes easy. In this respect, microkernel are superior to monolithic kernels. Other important issues are reliability, performance, and scalability.

REFERENCE

1. Avizienis A, and Kelly J. Fault Tolerance by Design Diversity. *Computer*, 1984, 17, 66-80
2. Birrell AD, Needham RM. A Universal File Server. *IEEE Trans. Software Eng.*, 1980, SE-6, 450-453
3. Birrell AD, Nelson BJ. Implementing Remote Procedure Calls. *ACM Trans. Comp. Syst.*, 1984, 2, 39-59
4. Dalal YK. Broadcast Protocols in Packet Switched Computer Networks., Ph. D. Thesis, Stanford Univ., 1977
5. Dennis JB, Van Horn EC. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 1966, 9, 143-154
6. Farber DJ, Larson KC. The System Architecture of the Distributed Computer System-The Communications System. *Symp. on Computer Netw.*, Polytechnic Institute of Brooklyn, April 1972
7. Fridrich M, Older W. The Felix File Server. *Proc. 8th Symp. on Operating Syst. Prin.*, ACM, 1981, 37-44
8. Lampson BW. Atomic Transactions in *Distributed Systems - Architecture and Implementation* , Berlin: Springer-Verlag, pp. 246-265, 1981
9. Ousterhout JK. Scheduling Techniques for Concurrent Systems. *Proc. 3rd Int'l Conf. on Distributed Computing Syst*, IEEE, 1982, 22-30
10. Powell ML, Presotto DL. Publishing-A Reliable Broadcast Communication Mechanism. *Proc. 9th Symp. on Operating Syst. Prin.*, ACM, 1983, 100-109
11. Sturgis HE, Mitchell JG, Israel J. Issues in the Design and Use of a Distributed File System. *Op. Syst. Rev.*, 1980, 14, 55-69
12. Swinehart D, McDaniel G, BoggsD. WFS: A Simple Shared File System for a Distributed Environment. *Proc. 7th Symp. on Operating Syst. Prin.*, ACM, 1979, 9-17
13. Van Tilborg AM, Wittie LD. Wave Scheduling: Distributed Allocation of Task Forces in Network Computers. *Proc. 2nd Int'l Conf. on Distributed Computing Syst.*, IEEE, 1981, 337-347
14. Wittie LD, van Tilborg AM. MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer. *IEEE Trans. Comp.*, 1980, 1133-1144
15. Zimmerman H. OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection. *IEEE Trans. Comm.*, 1980, 28, 425-432