NETWORK, NONLINEAR. See NONLINEAR NETWORK ELEMENTS.

NETWORK OPERATING SYSTEMS

Network operating systems extend the facilities and services provided by computer operating systems to support a set of computers, connected by a network. The environment managed by a network operating system consists of an interconnected group of machines that are loosely connected. By loosely connected, 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. Each computer in this group runs an autonomous operating system, yet cooperates with the others to allow a variety of facilities including file sharing, data sharing, peripheral sharing, remote execution, and cooperative computation. Network operating systems are autonomous operating systems that support such cooperation. The group of machines composing the management domain of the network operating system is called a *distributed system.* A close cousin of the network operating system is the distributed operating system. A distributed operating system is an extension of the network operating system that supports even higher levels of cooperation and integration of the machines on the network (features include task migration, dynamic resource location, and so on) (1,2).

An operating system is low-level software controlling the inner workings of a machine. Typical functions performed by an operating system include managing the CPU among many concurrently executing tasks, managing memory allocation to the tasks, handling of input and output, and controlling all the peripherals. Applications programs and often the human user are unaware of the existence of the features of operating systems as the features are embedded and hidden below many layers of software. Thus, the term *low-level software* is used. Operating systems were developed, in many forms, since the early 1960s and have matured in the 1970s. The emergence of networking in the 1970s and its explosive growth since the early 1980s have had a significant impact on the networking services provided by an operating system. As more network management features moved into the operating systems, network operating systems evolved.

Like regular operating systems, network operating systems provide services to the programs that run on top of the operating system. However, the type of services and the manner in which the services are provided are quite different. The services tend to be much more complex than those provided by regular operating systems. In addition, the implementation of these services requires the use of multiple machines, message passing, and server processes.

The set of typical services provided by a network operating system includes (but are not limited to):

1. Remote logon and file transfer
2. Transparent, remote file service
3. Directory and naming service
4. Remote procedure call service
5. Object and brokerage service

6. Time and synchronization service
7. Remote memory service

The network operating system is an extensible operating system. It provides mechanisms to easily add and remove services, reconfigure the resources, and has the ability of supporting multiple services of the same kind (for example two kinds of file systems). Such features make network operating systems indispensable in large networked environments.

In the early 1980s network operating systems were mainly research projects. Many network and distributed operating systems were built. These include such names as Amoeba, Argus, Berkeley Unix, Choices, Clouds, Cronus, Eden, Mach, Newcastle Connection, Sprite, and the V-System. Many of the ideas developed by these research projects have now moved into the commercial products. The commonly available network operating systems include Linux (freeware), Novell Netware, SunOS/Solaris, Unix, and Windows NT.

In addition to the software technology that goes into networked systems, theoretical foundations of distributed (or networked) systems have been developed. Such theory includes topics such as distributed algorithms, control of concurrency, state management, deadlock handling, and so on.

HISTORY

The emergence of and subsequent popularity of networking prompted the advent of network operating systems. The first networks supported some basic network protocol and allowed computers to exchange data. Specific application programs running on these machines controlled the exchange of data and used the network to share data for specific purposes. Soon it was apparent that a uniform and global networking support within the operating system would be necessary to effectively use the underlying network.

A particularly successful thrust at integrating networking extensions into an operating system resulted in Berkeley Unix (known as BSD). Unix was an operating system created at Bell Labs, and was licensed to the University of California at Berkeley for enhancements and then licensed quite freely to most universities and research facilities. The major innovation in Berkeley's version was support for TCP-IP networking.

In the early 1980s TCP-IP (or transmission control protocol-Internet protocol) was an emerging networking protocol, developed by a team of research institutions for a US Government funded project called the ARPANET. Specialized machines were connected to ARPANET and these machines ran TCP-IP. Berkeley made the groundbreaking decision to integrate the TCP-IP protocol into the Unix operating system, suddenly allowing all processes on a general-purpose Unix machine to communicate to other processes on any machine connected to the network. Then came the now ubiquitous programs that ran on top of the TCP-IP protocol. These programs include telnet, ftp, and e-mail.

The telnet program (as well as its cousins rlogin and rsh) allow a user on one machine to transparently access another machine. Similarly, ftp allows transmission of files between machines with ease. E-mail opened a new mode of communication.

While these facilities are very basic and taken for granted today, they were considered revolutionary when they first appeared. However, as the number of networked computers increased dramatically, it was apparent that these services were simply not enough for an effective work environment. For example, let us assume a department (in 1985) has about 40 users assigned to 10 machines. This assignment immediately led to a whole slew of problems, we outline some below:

- A user can only use the machine on which he or she has an account. Soon users started wanting accounts on many if not all machines.
- A user wanting to send mail to another colleague not only had to know the recipients name (acceptable) but which machines the recipient uses-in fact, the sender needs to know the recipient's favorite machine.
- Two users working together, but having different machine assignments have to use *ftp* to move files back and forth in order to accomplish joint work. This not only requires that they know each other's passwords but also they have to manually track the versions of the files.

Suddenly the boon of networking caused segregation of the workplace and became more of a bother rather than an enabling technology. At this point the systems designers realized the need for far tighter integration of networking and operating systems and the idea of a network operating system was born.

The first popular commercial network operating system was SunOS from Sun Microsystems. SunOS is a derivative from the popular Berkeley Unix (BSD). Two major innovations present in SunOS are called Sun-NFS and Yellow Pages. Sun-NSF is a network file system. Sun-NSF allows a file that exists on one machine to be transparently visible from other machines. Yellow Pages, which was later renamed to NIS (Network Information System), is a directory service. This service allowed, among other things, user accounts created in one central administrative machine to be propagated to machines the user needs to use.

The addition of better, global services to the base operating system is the basic concept that propelled the emergence of network operating systems. Current operating systems provide a rather large number of such services built at the kernel layer or at higher layers to provide application programs with a unified view of the network. In fact, the goal of network operating systems is network transparency; that is, the network becomes invisible to users and application programs.

SERVICES FOR NETWORK OPERATING SYSTEMS

System-wide services are the main facility a network operating system provides. These services come in many flavors and types. Services are functions provided by the operating system and form a substrate used by those applications, which need to interact beyond the simplistic boundaries imposed by the process concept.

A service is provided by a server and accessed by clients. A server is a process or task that continuously monitors incoming service requests (similar to telephone operators). When a service request comes in, the server process reacts to the request, performs the task requested, and then returns a response to the requestor. Often, one or more such server processes run on a computer and the computer is called a server.

What is a service? In regular operating systems, the system call interface or API (application programming interface) defines the set of services provided by the operating system. For example, operating system services include process creation facilities, file manipulation facilities, and so on. These services (or system calls) are predefined and static. However, this is not the case in a network operating system. Network operating systems do provide a set of static, predefined services, or system calls like the regular operating system, but in addition provides a much larger, richer set of dynamically creatable and configurable services. Additional services are added to the network operating system by the use of server processes and associated libraries.

Any process making a request to a server process is called a client. A client makes a request by sending a message to a server containing details of the request and awaiting a response. For each server, there is a well-defined protocol defining the requests that can be made to that server and the responses that are expected. In addition, any process can make a request; that is anyone can become a client, even temporarily. For example, a server process can obtain services from yet another server process, and while it is doing so, it can be termed a temporary client.

Services provided by a network operating system include file service, name service, object service, time service, and memory service.

Peripheral Sharing Service

Peripherals connected to one computer are often shared by other computers, by the use of peripheral sharing services. These services go by many names, such as remote device access, printer sharing, shared disks, and so on. A computer having a peripheral device makes it available by exporting it. Other computers can connect to the exported peripheral. After a connection is made, to a user on the machine connected to a shared peripheral, that peripheral appears to be local (that is, connected to the users machine). The sharing service is the most basic service provided by a network operating system.

File Service

The most common service that a network operating system provides is file service. File services allow a user of a set of computers to access files and other persistent storage objects from any computer connected to the network. The files are stored in one or more machines called the file server(s). The machines that use these files, often called *workstations,* have transparent access to these files.

Not only is the file service a common service, but it is also the most important service in the network operating system. Consequently, it is the most heavily studied and optimized service. There are many different, often noninteroperable protocols for providing file service (3).

The first full-fledged implementation of a file service system was done by Sun Microsystems and is called the Sun Network File System (Sun-NFS). Sun-NFS has become an industry standard network file system for computers running the Unix operating system. Sun-NFS can also be used from computers running Windows (all varieties) and MacOS but with some limitations.

Under Sun-NFS a machine on a network can export a file system tree (i.e. a directory and all its contents and subdirectories). A machine that exports one of more directories is called a file server. After a directory has been exported, any machine connected to the file server (could be connected over the Internet) can import, or mount that file tree. Mounting is a process, by which the exported directory, all its contents, and all its subdirectories appear to be a local directory on the machine that mounted it. Mounting is a common method used in Unix system to build unified file systems from a set of disk partitions. The mounting of one exported directory from one machine to a local directory on another machine via Sun-NFS is termed *remote mounting.*

Figure 1 shows two file servers, each exporting a directory containing many directories and files. These two exported directories are mounted on a set of workstations, each workstation mounting both the exported directories from each of the file servers. This configuration results in a uniform file space structure at each the workstation.

While many different configurations are possible by the innovative use of remote mounting, the system configuration shown in Fig. 1 is quite commonly used. This is called the *dataless* workstation configuration. In such a setup, all files, data and critical applications are kept on the file servers and mounted on the workstations. The local disks of the workstations only contain the operating system, some heavily used applications and swap space.

Sun-NFS works by using a protocol defined for remote file service. When an application program makes a request to read (or write) a file, it makes a local system call to the operating system. The operating system then consults its mounting tables to determine if the file is a local file or a remote file. If the file is local, the conventional file access mechanisms handle the task. If the file is remote, the operating system creates a request packet confirming to the NFS protocol and sends the packet to the machine having the file.

The remote machine runs a server process, also called a *daemon,* named nfsd. Nfsd receives the request and reads (or

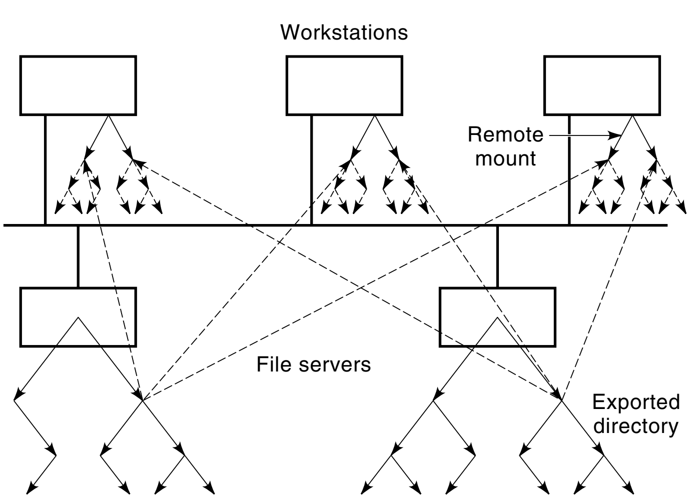

Figure 1. The file mounting structure for Sun NFS.

writes) the file, as requested by the application and returns a confirmation to the requesting machine. Then, the requesting machine informs the application of the success of the operation. Of course, the application does not know whether the execution of the file operation was local or remote.

Similar to Sun-NFS, there are several other protocols for file service. These include Appleshare for Macintosh computers, the SMB protocol for Windows 95/NT, and the DFS protocol used in the Andrew file system. Of these, the Andrew file system is the most innovative.

Andrew, developed at CMU in the late 1980s, is a scalable file system. Andrew is designed to handle hundreds of file servers and many thousands of workstations without degrading the file service performance. Degraded performance in other file systems is the result of bottlenecks at file servers and network access points. The key feature that makes Andrew a scalable system is the use of innovative file caching strategies. A file being used at a workstation is cached (i.e., a copy is kept) in its entirety at the workstation or at an intermediate server close to the workstation. Updates are applied to the cached copy, and later transmitted to the file server. The Andrew file system is also available commercially and is called DFS (distributed file system).

In Andrew/DFS when an application accesses a file, the entire file is transmitted from the server to the workstation, or a special intermediate file storage system, closer to the workstation. Then, the application uses the file, in a manner similar to NFS. After the user running the application logs out of the workstation, the file is sent back to the server. Such a system however has the potential of suffering from file inconsistencies if the same user uses two workstations at two locations.

In order to keep files consistent, when it is used concurrently, the file server uses a callback protocol. The server can recall the file in use by a workstation if another workstation uses it simultaneously. Under the callback scheme, the server stores the file and both workstations reach the file remotely. Performance suffers, but consistency is retained. Since concurrent access to a file is rare, the callback protocol is very infrequently used; and thus does not hamper the scalability of the system.

Directory or Name Service

A network of computers managed by a network operating system can get rather large. A particular problem in large networks is the maintenance of information about the availability of services and their physical location. For example, a particular client needs access to a database. There are many different database services running on the network. How would the client know whether the particular service it is interested in, is available, and if so, on what server?

Directory services, sometimes called *name services,* address such problems. Directory services are the mainstay of large network operating systems. When a client application needs to access a server process, it contacts the directory server and requests the address of the service. The directory server identifies the service by its name-all services have unique names. Then, the directory server informs the client of the address of the service-the address contains the name of the server. The directory server is responsible for knowing the current locations and availability of all services and hence can inform the client of the unique network address (somewhat like a telephone number) of the service.

The directory service is thus a database of service names and service addresses. All servers register themselves with the directory service upon startup. Clients find server addresses upon startup. Clients can retain the results of a directory lookup for the duration of its life, or can store it in a file and thus retain it potentially forever. Retaining addresses of services is termed *address caching.* Address caching causes gains in performance and reduces loads on the directory server. Caching also has disadvantages. If the system is reconfigured and the service address changes, then the cached data is wrong and can indeed cause serious disruptions if some other service is assigned that address. Thus, when caching is used, clients and servers have to verify the accuracy of cached information.

The directory service is just like any other service, i.e. it is provided by a service process. So there are two problems:

1. How does the client find the address of the directory service?
2. What happens if the directory service process crashes?

Making the address of the directory service a constant solves the first problem. Different systems have different techniques for doing this, but a client always has enough information about contacting the directory service.

To ensure the directory service is robust and not dependent on one machine, the directory service is often replicated or mirrored. That is, there are several independent directory servers and all of them contain (hopefully) the same information. A client is aware of all these services and contacts any one. As long as one directory service is reachable, the client gets the information it seeks. However, keeping the directory servers consistent, i.e. having the same information, is not a simple task. This is generally done by using one of many replication control protocols (see section entitled "Theoretical Foundations").

The directory service has been subsequently expanded not just to handle service addresses, but higher level information such as user information, object information, and web information. A standard for worldwide directory services over large networks such as the Internet has been developed and is known as the X.500 directory service. However the deployment of X.500 has been low and thus its importance has eroded. A simpler directory service called LDAP (lightweight directory access protocol) is more popular, and most network operating systems provide support for this protocol.

RPC Service

A particular mechanism for implementing the services in a network operating system is called remote procedure calls or RPC. The RPC mechanism is discussed later in the section entitled "Mechanisms for Network Operating Systems." The RPC mechanism needs the availability of an RPC server accessible by an RPC client. However, a particular system may contain tens if not hundreds or even thousands of RPC servers. In order to avoid conflicts and divergent communication protocols the network operating system provides support for building and managing and accessing RPC servers.

Each RPC service is an application-defined service. However, the operating system also provides an RPC service, which is a meta-service, which allows the application-specific RPC services to be used in a uniform manner. This service provides several features:

1. Management of unique identifiers (or addresses) for each RPC server.
2. Tools for building client and server stubs for packing and unpacking (also known as marshalling and unmarshalling) of arguments between clients and servers.
3. A per-machine RPC listening service.

The RPC service defines a set of unique numbers that can be used by all RPC servers on the network. Each specific RPC server is assigned one of these numbers (addresses). The operating system manages the creation and assignment of these identifiers. The operating system also provides tools that allow the programmers of RPC services to build a consistent client–server interface. This is done by the use of language processing tools and stub generators, which embed routines in the client and server code. These routines package the data sent from the client to the server (and vice versa) in some predefined format, which is also machine independent.

When a client uses the number to contact the service, its looks up the directory and finds the name of the physical machine that contains the service. Then it sends a RPC request to the RPC listener on that machine. The RPC listener is an operating-system-provided service that redirects RPC calls to the actual RPC server process that should handle the call.

RPC services are available in all network operating systems. The three most common types of RPC systems are Sun RPC, DCE RPC, and Microsoft RPC.

Object and Brokerage Service

The success and popularity of RPC services coupled with the object-orientation frenzy of the mid-1980s led to the development of object services and then to brokerage services. The concept of object services is as follows.

Services in networked environments can be thought of as basic services and composite services. Each basic service is implemented by an object. An object is an instance of a class, while a class is inherited from one or more base or composite classes. The object is a persistent entity that stores data in a structured form, and may contain other objects. The object has an external interface, visible from clients and is defined by the public methods the object supports.

Composite services are composed of multiple objects (basic and composite) which can be embedded or linked. Thus, we can build a highly structured service infrastructure that is flexible, modular, and has unlimited growth potential.

In order to achieve this concept, the network operating systems started providing uniform methods of describing, implementing, and supporting objects (similar to the support for RPC).

While the concept sounds very attractive in theory, there are some practical problems. These are:

1. How does a client access a service?
2. How does a client know of the available services and the interfaces they offer?
3. How does one actually build objects (or services)?

We discuss the questions in reverse order. The services or objects are built using a language that allows the specification of objects, classes, and methods, and allows for inheritance and overloading. While C++ seems to be a natural choice, C++ does not provide the features of defining external service interfaces and does not have the power of remote linking. Therefore, languages have been defined based on C++ that provide such features.

The client knows of the object interface, due to the predefined type of the object providing the service. The programming language provides and enforces the type information. Hence at compile time, the client can be configured by the compiler to use the correct interface based on the class of object the client is using. However, such a scheme makes the client use a *static* interface. That is, once a client has been complied the service cannot be updated with new features that change the interface. This need for dynamic interface management leads to the need for brokerage services.

After the client knows of the existence of the service, and the interface it offers, the client accesses the service using two key mechanisms—the client stub and the ORB (object request broker). The client stub transforms a method invocation into a transmittable service request. Embedded in the service request is ample information about the type of service requested and the arguments (and type of these arguments) and the type of expected results. The client stub then sends a message to the ORB handling requests of this type.

The ORB is just one of the many services a brokerage system provides. The ORB is responsible for handling client requests and is an intermediate between the client and the object. Thus, the ORB is a server-side stub that receives incoming service requests and converts them to correct formats, and sends them to the appropriate objects.

The Brokerage Service is a significantly more complex entity. It is responsible for handling:

1. Names and types of objects and their locations and types.
2. Controlling the concurrency of method invocations on objects, if they happen concurrently.
3. Event notification and error handling.
4. Managing the creation and deletion of objects and updates of objects as they happen, dynamically.
5. Handling the persistence and consistency of objects. Some critical objects may need transaction management.
6. Handling queries about object capabilities and interfaces.
7. Handling reliability and replication.
8. Providing trader services.

The trader service adds more features to the object services. The main power in object services is unleashed when clients can pick and choose services dynamically. For example, a client wants access to a database object containing movies. Many such services may exist on the network offering different or even similar features. The client can first contact the trader, get information about services (including quality, price, range of offerings, and so on) and then decide to use one of them. This is, of course, based on the successful, real-world business model. Trader services thus offer viable and useful methods of interfacing clients and objects on a large network.

The object and brokerage services depend heavily on standards, as all programs running on a network have to conform to the same standard, in order to interoperate. As of writing, the OSF-DCE (open software foundation, distributed computing environment) is the oldest multiplatform standard, but has limited features (does not support inheritance, dynamic interfaces, and so on). The CORBA (common object request broker architecture) standard is gaining importance as a much better standard and is being deployed quite aggressively. Its competition, the DCOM (distributed common object model) standard is also gaining momentum, but its availability seems to be currently limited to the Windows family of operating systems.

Group Communication Service

Group communication is an extension of multicasting for communicating process groups. When the recipient of a message is a set of processes the message is called a *multicast* message (a single recipient message—unicast, all processes are recipients—broadcast). A process group is a set of processes whose membership may change over time. If a process sends a multicast message to a process group, all processes that are members of the group will receive this message. Simple implementations of multicasting does not work for group communications for a variety of reasons, such as:

1. A process may leave the group and then get messages sent to the group from a process who is not yet aware of the membership change.
2. Process P_1 sends a multicast. In response to the multicast, process P_2 sends another multicast. However, P_2's message arrives at P_3 before P_1's message. This is causally inconsistent.
3. Some processes, which are members of the group, may not receive a multicast due to message loss or corruption.

The main provision in a group communication system is the provision of multicasting primitives. Some of the important ones are:

Reliable Multicast. The multicast is send to all processes and then retransmitted to processes that did not get the message, until all processes get the multicast. Reliable multicasts may not deliver all messages if some network problems arise.

Atomic Multicast. Similar to the reliable multicast, but guarantees that all processes will receive the message. If it is not possible for all processes to receive the message, then no process will receive the message.

Totally Ordered Multicast. All the multicasts are ordered strictly; that is, all the receivers get all the messages in exactly the same order. Totally ordered multicasting is expensive to implement and is not necessary (in most cases). Causal multicasting is powerful enough for use by applications that need ordered multicasting.

Causally Ordered Multicast. If two multicast messages are causally related in some way then all recipients of these multicasts will get them in the correct order.

Imperative in the notion of multicasting is the notion of dynamic process groups. A multicast is sent to a process group and all current members of that group receive the message. The sender does not have to belong to the group.

Group communication is especially useful in building fault-tolerant services. For example, a set of separate servers providing the same service are assigned to a group and all service requests are sent via causally ordered multicasting. Now all the servers will do exactly the same thing, and if one server fails, it can be removed from the group. This approach is used in the ISIS system (4).

Time, Memory, and Locking Services

Managing time on a distributed system is inherently conceptually difficult. Each machine runs its own clock and these clocks drift independently. In fact there is no method to even initially synchronize the clocks. Time servers provide a notion of time to any program interested in time, based on one of many clock algorithms (see section on theoretical foundations). Time services have two functions: provide consistent time information to all processes on the system and to provide a clock synchronization method that ensures all clocks on all systems appear to be logically synchronized.

Memory services provide a logically shared memory segment to processes not running on the same machine. The method used for this service is described later. A shared memory server provides the service, and processes can attach to a shared memory segment which is automatically kept consistent by the server.

There is often a need for locking a resource on the network, by a process. This is especially true in systems using shared memory. While locking is quite common and simple in single computers, it is not so easy on a network. Thus, networks use a locking service. A locking service is typically a single server process that tracks all locked resources. When a process asks for a lock on a resource, the server grants the lock if that lock is currently not in use, or else it makes the requesting process wait until the lock is released.

Other Services

A plethora of other services exists in network operating systems. These services can be loosely divided into two classes: (1) services provided by the core network operating system and (2) services provided by applications.

Services provided by the operating system are generally low-level services used by the operating system itself, or by applications. These services of course vary from one operating system to another. The following is a brief overview of services provided by most operating systems that use the TCP-IP protocol for network communications:

1. *Logon Services.* These include telnet, rlogin, ftp, rsh, and other authentication services that allow users on one machine to access facilities of other machines.
2. *Mail Services.* These include SMTP (simple mail transfer protocol), POP (post office protocol), and IMAP (Internet message access protocol). These services provide the underlying framework for transmitting and accessing electronic mail. The mail application provides a nicer interface to the end user, but uses several of these low-level protocols to actually transmit and receive mail messages.
3. *User Services.* These include finger, rwho, whois, and talk.
4. *Publishing Services.* These include HTTP (hyper-text transfer protocol), NNTP (network news transfer protocol), Gopher, and WAIS. These protocols provide the backbone of the Internet information services such as the WWW and the news network.

Application-defined services, on the other hand, are used by specific applications that run on the network operating system. One of the major attributes of a network operating system is that it can provide support for distributed applications. These application programs span machine boundaries and user boundaries. That is, these applications use resources (both hardware and software) of multiple machines and input from multiple users to perform a complex task. Examples include parallel processing and CSCW (computer supported cooperative work).

Such distributed applications use the RPC services or object services provided by the underlying system to build services specific to the type of computation being performed. Parallel processing systems use the message passing and RPC mechanisms to provide remote job spawning and distribution of computational workload among all available machines on the network. CSCW applications provide services such as whiteboards and shared workspaces, which can be used by multiple persons at different locations on the network.

A particular, easy to understand application is a calendering program. In calendaring applications, a server maintains information about appointments and free periods of a set of people. All individuals set up their own schedules using a front-end program, which downloads such data into a server. If a person wants to set up a meeting, he or she can query the server for a list of free periods, for a specified set of people. After the server provides some alternatives, the person schedules a particular time and informs all the participants. While the scheduling decision is pending, the server marks the appointment time temporarily unavailable on the calendars of all participating members. Thus, the calendering application provides its own unique service—the calendar server.

MECHANISMS FOR NETWORK OPERATING SYSTEMS

Network operating systems provide three basic mechanisms that are used to the support the services provided by the operating system and applications. These mechanisms are (1) message passing, (2) remote procedure calls (RPC), and (3) distributed shared memory (DSM). These mechanisms support a feature called *interprocess communication* or IPC. While all the mechanisms are suitable for all kinds of interprocess communication, RPC and DSM are favored over message passing by programmers.

Message Passing

Message passing is the most basic mechanism provided by the operating system. This mechanism allows a process on one machine to send a packet of raw, uninterpreted stream of bytes to another process.

In order to use the message passing system, a process wanting to receive messages (or the receiving process) creates a port (or mailbox). A port is an abstraction for a buffer, in which incoming messages are stored. Each port has a unique system-wide address, which is assigned, when the port is created. A port is created by the operating system upon a request from the receiving process and is created at the machine where the receiving process executes. Then, the receiving process may choose to register the port address with a directory service.

After a port is created, the receiving process can request the operating system to retrieve a message from the port and provide the received data to the process. This is done via a receive system call. If there are no messages in the port, the process is blocked by the operating system until a message arrives. When a message arrives, the process is woken up and is allowed to access the message.

A message arrives at a port, after a process sends a message to that port. The sending process creates the data to be sent and packages the data in a packet. Then it requests the operating system to deliver this message to the particular port, using the address of the port. The port can be on the same machine as the sender, or a machine connected to the same network.

When a message is sent to a port that is not on the same machine as the sender (the most common case) this message traverses a network. The actual transmission of the message uses a networking protocol that provides routing, reliability, accuracy, and safe delivery. The most common networking protocol is TCP-IP. Other protocols include IPX/SPX, AppleTalk, NetBEUI, and PPTP. Network protocols use techniques such as packetizing, checksums, acknowledgements, gatewaying, routing, and flow control to ensure messages that are sent are received correctly and in the order they were sent.

Message passing is the basic building block of distributed systems. Network operating system use message passing for interkernel as well as interprocess communications. Interkernel communications are necessary as the operating system on one machine needs to cooperate with operating systems on other machines to authenticate users, manage files, handle replication, and so on.

Programming using message passing is achieved by using the send/receive system calls and the port creation and registering facilities. These facilities are part of the message passing API provided by the operating system. However, programming using message passing is considered to be a low-level technique that is error prone and best avoided. This is due to the unstructured nature of message passing.

Message passing is unstructured, as there are no structural restrictions on its usage. Any process can send a message to any port. A process may send messages to a process that is not expecting any. A process may wait for messages from another process, and no message may originate from the second process. Such situations can lead to bugs that are very difficult to detect. Sometimes timeouts are used to get out of the blocked receive calls when no messages arrive, but the message may actually arrive just after the timeout fires.

Even worse, the messages contain raw data. Suppose a sender sends three integers to a receiver who is expecting one floating-point value. This will cause very strange and often undetected behaviors in the programs. Such errors occur frequently due to the complex nature of message passing programs and hence better mechanisms have been developed for programs that need to cooperate.

Even so, a majority of the software developed for providing services and applications in networked environments uses message passing. Some minimization of errors is done by strictly adhering to a programming style called the *client–server programming paradigm.* In this paradigm, some processes are predesignated as servers. A server process consists of an infinite loop. Inside the loop is a receive statement which waits for messages to arrive at a port called the service port. When a message arrives, the server performs some task requested by the message and then executes a send call to send back results to the requestor and goes back to listening for new messages.

The other processes are clients. These processes send a message to a server and then wait for a response using a *receive.* In other words, all *sends* in a client process must be followed by a *receive* and all *receives* at a server process must be followed by a *send.* Following this scheme significantly reduced timing related bugs.

The performance of client–server-based programs are, however, poorer than what can be achieved by raw message passing. To alleviate this, often a multithreaded server is used. In a multithreaded server several parallel threads can listen to the same port for incoming messages and perform requests in parallel. This causes quicker service response times. Two better interprocess communication techniques are RPC and DSM.

Remote Procedure Calls

RPC is a method of performing interprocess communication with a familiar, procedure-call-like mechanism. In this scheme, to access remote services, a client makes a procedure call, just like a regular procedure call, but the procedure executes within the context of a different process, possibly on a different machine. The RPC mechanism is similar to the client–server programming style used in message passing. However, unlike message passing where the programmer is responsible for writing all the communication code, in RPC a compiler automates much of the intricate details of the communication.

In concept, RPC works as follows: A client process wishes to get service from a server. It makes a remote procedure call on a procedure defined in the server. In order to do this the client sends a message to the RPC listening service on the machine where the remote procedure is stored. In the message, the client sends all the parameters needed to perform the task. The RPC listener then activates the procedure in the proper context, lets it run, and returns the results generated by the procedure to the client program. However, much of this task is automated and not under programmer control.

An RPC service is created by a programmer who (let us assume) writes the server program as well as the client program. In order to do this, he or she first writes an interface description using a special language called the *interface description language* (IDL). All RPC systems provide an IDL definition and an IDL compiler. The interface specification of a server documents all the procedures available in the server and the types of arguments they take and the results they provide.

The IDL compiler compiles this specification into two files, one containing C code that is to be used for writing the server program and the other containing code used to write the client program.

The part for the server contains the definitions (or prototypes) of the procedures supported by the server. It also contains some code called the *server loop.* To this template, the programmer adds the global variables, private functions, and the implementation of the procedures supported by the interface. When the resulting program is compiled, a server is generated. The server loop is inserted by the IDL compiler contains code to:

1. Register the service with a name server.
2. Listen for incoming requests (could be via the listening service provided by the operating system).
3. Parse the incoming request and call the appropriate procedure using the supplied parameters. This step requires the extraction of the parameters from the message sent by the client. The extraction process is called *unmarshalling.* During unmarshalling some type checking can also be performed.
4. After the procedure returns, the server loop packages the return results into a message (marshalling) and sends a reply message to the client.

Note that all of this functionality is automatically inserted into the RPC server by the IDL compiler and the programmer does not have to write any of these.

Then, the programmer writes the client. In the client program, the programmer includes the header file for clients generated by the IDL compiler. This file has the definitions and pseudo-implementations (or proxies) of the procedures that are actually in the server. The client program is written as if the calls to the remote procedures are in fact local procedure calls. When the client program is run, the stubs inserted via the header files play an important role in the execution of the RPCs.

When the client process makes a call to a remote procedure, it actually calls a local procedure, which is a proxy for the remote procedure. This proxy procedure (or stub) gets all the arguments passed to it and packages them in some predefined format. This packaging is called *marshalling.* After the arguments are marshaled, they are sent to the RPC server that handles requests for this procedure. Of course, as described, the RPC server unmarshals arguments, runs the procedure, and marshals results. The results flow back to the client, and the proxy procedure gets them. It unmarshals the results and returns control to the calling statement, just like a regular local procedure.

One problem remains. How does the client know what is the address of the server handling a particular procedure call? This function is automated too. The IDL compiler, when compiling an interface definition, obtains a unique number from the operating system and inserts it into both the client stub and the server stub, as a constant. The server registers this number with its address on the name service. The client uses this number to look up the server's address from the name service.

The net effect is that a programmer can write a set of server routines, which can be used from multiple client processes running on a network of machines. The writing of these routines takes minimal effort and calling them from remote processes is not difficult either. There is no need to write communications routines and routines to manage arguments and handle type checking. Automation reduces chances of bugs quite heavily. This has led to the acceptance of RPC as the preferred distributed programming tool.

Distributed Shared Memory

While message passing and RPC are the mainstays of distributed programming, and are available on all network operating systems, DSM is not at all ubiquitous. On a distributed system, DSM provides a logical equivalent to (real) shared memory, which is normally available only on multiprocessor systems.

Multiprocessor systems have the ability of providing the same physical memory to multiple processors. This is a very useful feature and has been utilized heavily for parallel processing and interprocess communication in multiprocessor machines. While RPC and message passing is also possible on multiprocessor systems, using shared memory for communication and data sharing is more natural and is preferred by most programmers.

While shared memory is naturally available in multiprocessors, due to the physical design of the computer, it is neither available nor is thought to be possible on a distributed system. However, the DSM concept has proven that a logical version of shared memory, which works just like the physical version, albeit at reduced performance, is both possible and is quite useful.

DSM is a feature by which two or more processes on two or more machines can map a single shared memory segment to their address spaces. This shared segment behaves like real shared memory; that is, any change made by any process to any byte in the shared segment is instantaneously seen by all the processes that map the segment. Of course, this segment cannot be at all the machines at the same time, and updates cannot be immediately propagated, due to the limitations of speed of the network.

DSM is implemented by having a DSM server that stores the shared segment; that is, it has the data contained by shared segment. The segment is an integral number of pages. When a process maps the segment to its address space, the operating system reserves the address range in memory and marks the virtual addresses of the mapped pages as inaccessible (via the page table). If this process accesses any page in the shared segment, a page fault is caused. The DSM client is the page fault handler of the process.

The workings of DSM are rather complex due to the enormous number of cases the algorithm has to handle. Modern DSM systems provide intricate optimizations that make the system run faster but are hard to understand. In this section, we discuss a simple, un-optimized DSM system, which if implemented would work, but would be rather inefficient.

DSM works with memory by organizing it as pages (similar to virtual memory systems). The mapped segment is a set of pages. The protection attributes of these pages are set to inaccessible, read only, or read-write:

1. *Inaccessible.* This denotes that the current version of the page is *not* available on this machine and the server needs to be contacted before the page can be read or written.
2. *Read-only.* This denotes that the most recent version of the page is available on this machine; i.e., the process on this machine holds the page in read mode. Other processes may also have the page in read-only mode, but no process has it in write mode. This page can be freely read, but not updated without informing the DSM server.
3. *Read-write.* This denotes that this machine has the sole, latest version of the page; i.e., the process on this machine holds the page in write mode. No other process has a copy of this page. It can be freely read or updated. However, if this page is needed anywhere else, the DSM server may yank the privileges by invalidating the page.

The DSM client or page fault handler is activated whenever there is a page fault. When activated, the DSM client first determines whether the page fault was due to a read access or a write access. The two cases are different and are described separately.

Read Access Fault. On a read access fault, the DSM client contacts the DSM server and asks for the page in read mode. If there are no clients that have already requested the page in write mode, the server sends the page to the DSM client. After getting the page, the DSM client copies it into the memory of the process, at the correct address, and sets the protection of the page as read only. It then restarts the process that caused the page fault.

If there is one client already holding the page in write mode (there can be at most one client in write mode) then the server first asks the client to relinquish the page. This is called invalidation. The client relinquishes the page by sending it back to the server and marking the page as inaccessible. After the invalidation is done, the server sends the page to the requesting client, as before.

Write Access Fault. On a write access fault, the DSM client contacts the server and requests the page in write mode. If the page is not currently used in read or write mode by any other process, the server provides a copy of the page to the client. The client then copies the page to memory, sets the protection to read-write, and restarts the process.

If the page is currently held by some processes in read or write mode, the server invalidates all these copies of the page. Then, it sends the page to the requesting client, which installs it and sets the protection to read-write.

The net effects of this algorithm is as follows:

1. Only pages that are used by a process on a machine migrate to that machine.
2. Pages that are read by several processes migrate to the machines these processes are running on. Each machine has a copy.
3. Pages that are being updated migrate to the machines they are being updated on; however, there is at most one update copy of the page at any point in time. If the page is being simultaneously read and updated by two or more machines, then the page shuttles back and forth between these machines.

Page shuttling is a serious problem in DSM systems. There are many algorithms used to prevent page shuttling. Effective page shuttling prevention is done by relaxed memory coherence requirements, such as release consistency. Also, with careful design of applications page shuttling can be minimized.

The first system to incorporate DSM was Ivy (5). Several DSM packages are available; these include TreadMarks, Quarks, Avalanche, and Calypso.

KERNEL ARCHITECTURES

Operating systems always have been constructed (and often still are) using the monolithic kernel approach. The monolithic kernel is a large piece of protected software that implements all the services the operating system has to offer via a system call interface (or API). This approach has some significant disadvantages. The kernel, unlike application programs, is not a sequential program. A kernel is an interrupt driven program. That is, different parts of the kernel are triggered and made to execute at different (and unpredictable) points in time, due to interrupts. In fact, the entire kernel is interrupt driven. The net effect of this structure is that:

1. The kernel is hard to program. The dependencies of the independently interrupt-triggerable parts are hard to keep track of.
2. The kernel is hard to debug. There is no way of systematically running and testing the kernel. When a kernel is deployed, random parts start executing quite unpredictably.
3. The kernel is crucial. A bug in the kernel causes applications to crash, often mysteriously.
4. The kernel is very timing dependent. Timing errors are very hard to catch problems that are not repeatable and the kernel often contains many such glitches that are not detectable.

The emergence of network operating systems saw the sudden drastic increase in the size of kernels. This is due to the addition of a whole slew of facilities in the kernel, such as message passing, protocol handling, network device handling, network file systems, naming systems, RPC handling, and time management. Soon it was apparent that this bloat led to kernel implementations that are unwieldy, buggy, and doomed to fail.

This rise in complexity resulted in the development of an innovative kernel architecture targeted at network operating systems, called the *microkernel architecture.* A true microkernel places only those features in the kernel that positively have to be in the kernel. This includes low-level service such as CPU scheduling, memory management, device drivers, and network drivers. Then, it places a low-level message passing interface in the kernel. The user-level API is just essentially the message passing routines.

All other services are built outside the kernel, using server processes. It has been shown that almost every API service and all networking services can be placed outside the kernel. This architecture has some significant benefits, a few of which are listed:

1. Services can be programmed and tested separately. Changes to the service do not need recompiling the microkernel.
2. All services are insulated from each other—bugs in one service do not affect another service. This is not only a good feature, but makes debugging significantly easier.
3. Adding, updating, and reconfiguring services are easy.
4. Many different implementations of the same service can coexist.

Microkernel operating systems that proved successful include Amoeba (6), Mach (7), and the V-System (8). A commercial microkernel operating system called Chorus is marketed by Chorus Systems (France).

The advantages of microkernels come at a price, namely performance. Performance of operating systems is an all-important feature that can make or break the usage of the system, especially commercial systems. Hence, commercial systems typically shun the microkernel approach but choose a compromise called the *hybrid kernel.* A hybrid kernel is a microkernel in spirit, but a monolithic kernel in reality. The Chorus operating system pioneered the hybrid kernel. Windows NT is also a hybrid system.

A hybrid system starts as a microkernel. Then, as services are developed and debugged they are migrated into the kernel. This retains some of the advantages of the microkernel, but the migration of services into the kernel significantly improves the performance.

THEORETICAL FOUNDATIONS

The theoretical study of autonomous but networked computing system was propelled by the need for algorithms for use in networked environments. This active field of research has produced some interesting and seminal results. Much of the foundational work has resulted in the development of distributed algorithms (9). These algorithms are designed to allow a set of independent processes, running on independent computers (or machines, or nodes) to cooperate and interact to achieve a common goal. Many such algorithms are used for application programming. Some of the algorithms are, however, relevant to management of distributed systems and are used in network operating systems. In the following sections, we present a few algorithms which form the theoretical foundations of network and distributed operating systems. These include time management, deadlock handling, mutual exclusion, checkpointing, deadlocks detection, concurrency control, consensus, and replication control.

Distributed Clocks

Each physical machine on a network has its own clock, which is a hardware counter. This clock runs freely, and cannot be physically synchronized with other clocks. This makes the notion of time on a distributed system hard to define and obtain. The first clock synchronization algorithm provided a method of logically synchronizing clocks such that no application running on the system could ever detect any drift amongst the physical clocks (even though the clocks do drift). Clocks on systems built using this technique are called *Lamport clocks* after the inventor of the algorithm (10).

The Lamport clock algorithm works by stamping a time on every message outgoing from any machine. When the operating system on system S_i sends out a message, it stamps it with the time T_i, where T_i is the time according to the physical clock on S_i.

Suppose the message is received by the operating system on system S_j. The operating system on S_j checks the timestamp in the message with the time according to the local clock on S_j, i.e. T_j:

- If $T_i < T_j$ then no action is needed.
- If $T_i > T_j$ then the clock on S_j is incremented to T_{i+1}.

This action, at the least, ensures that no messages are received *before* they are sent. However, it also has some interesting side effects. These are:

- All clocks follow the fastest clock.
- The clocks are not physically synchronized, but they are logically synchronized. That is, to all applications running on the systems, the clocks appear completely synchronized.
- If two actions or events on two different machines are transitively related; that is, there is a chain of events from the occurrence of event i to the occurrence of event j; then the time of occurrence of i will always be lower than the time of occurrence of j. Even if i and j happened on two different machines with two different clocks.

The Lamport clock is a very simple algorithm which produces properly synchronized (logical) distributed clocks. However, it has the shortcoming that clocks cannot be set back, and hence real time clocks cannot use this method. In fact, setting back a clock will cause it to race ahead to catch up with the fastest clock. This problem is solved by the use of vector clocks.

In the vector clock scheme, each system clock is independent and is never updated by the clock algorithm. Every system maintains its own time, and information about the time on other systems. That is, there is a local clock on each system, as well as registers containing some approximation of the time on the sibling systems.

The time is maintained as an n-tuple (or vector) where n is the number of systems on the network. Each machine maintains this n-tuple. On machine S_i, the n-tuple (or the time vector) is T_n. T_n, of course has n fields and $T_n(i)$ is the local clock time. The other fields are updated in accordance to the following algorithm.

When a message is sent from S_i to S_j, the value of T_i is sent along with the message. When S_j receives the message, it updates its time vector T_j by updating each field in T_j to the larger of the values contained in the corresponding fields of T_i and T_j.

Now it can be shown that any two timestamps can be compared using vector clock algebra. Suppose we want to compare two timestamps T_a and T_b. Each has n fields, $T_{a(0)}$ to $T_{a(n-1)}$. The comparison operators are defined below.

Equal: For all i, $T_{a(i)}$ is equal to $T_{b(i)}$.

Not equal: For some i $T_{a(i)}$ is not equal to $T_{b(i)}$.

Less than or equal: For all i, $T_{a(i)}$ is less than or equal to $T_{b(i)}$.

Not less than or equal: For some i, $T_{a(i)}$ is not less than or equal $T_{b(i)}$.

Less than: T_a is less than or equal to T_b, and T_a is not equal T_b.

Concurrent: Not T_a less than T_b and not T_b less than T_a.

The vector clock thus provides all the functions of Lamport clocks as far as timestamps and event ordering is concerned. It is also just as simple to implement, but the time on one machine can be adjusted without affecting the time on other machines.

Distributed Mutual Exclusion

Distributed mutual exclusion (DME) is a classic problem in distributed computing. There are n processes executing on n sites. Each process is an infinite loop and has a critical section inside the loop. How do you ensure that at most one process executes within its critical section at any given time?

The easy solution is to use a lock server. Each process asks the lock server for permission to enter. The lock server permits only one process at a time. When a process leaves the critical section, it informs the lock server and the lock server can now allow another process to enter. This solution is called the *centralized solution* to the DME problem.

This solution is called centralized because all decisions are made at one site. In a problem such as DME, we can define two sets for each site. A site i has a request set Q_i and a response set R_i. Q_i is the set of sites that i will contact when it wants to enter the critical section. R_i is the set of sites that contact i if they want to enter the critical section (11).

In order for a mutual exclusion algorithm to be distributed, two rules must apply. These are:

Equal responsibility rule: For all i, j, $|Q_i| = |Q_j|$.

Equal effort rule: For all i, j, $|R_i| = |R_j|$.

In the centralized case, R_i for all i is the lock server site; and for all i, Q_i is empty. Thus, the centralized solution fails the two rules. Many different DME algorithms can meet such rules. Lamport proposed the first solution. In the Lamport algorithm, there are three steps:

Step 1: When a process wants to enter the critical section, it sends a request message, along with a timestamp, to all other processes, including itself. Upon receiving such a message, each process queues the request in timestamp order in a local request queue and sends an acknowledgment. The requesting process waits for all acknowledgments before proceeding.

Step 2: A process can enter when it notices that its own request is the first request in its own local request queue.

Step 3: Whenever a process exits the critical section it informs all processes and they remove the exiting processes request from their local request queues.

This algorithm meets the equal responsibility and equal effort rules. It uses $3n$ messages per entry into a critical section. The number of messages can be reduced to sqrt(n) by using a type of algorithm first proposed by Maekawa (11). Currently there are a large number of algorithms each having some advantage over the other.

Note that in most practical situations, the centralized algorithm works better and uses the lowest number of messages (just 2 messages per entrance). Thus, it is the most commonly used algorithm.

Distributed Checkpoints

Checkpointing is a method used to restart or debug computations. On a centralized operating system, checkpointing is easy, the process to be checkpointed is stopped and its memory contents are written to a file, then the process can continue execution. The checkpoint can later be used to restart the process (in case of failure) or to analyze its execution (in case of debugging).

In a networked or distributed system this technique does not work. Consider two processes P_1 and P_2. P_1 sends a message to P_2. We ask both P_1 and P_2 to stop and checkpoint themselves. P_1 does so, and then continues, and sends a message to P_2. P_2 receives the message from P_1 and then receives the checkpoint notification and then checkpoints itself. Now, if we compare the checkpoints of P_1 and P_2, we find P_2 has received a message that has not yet been sent by P_1. This is called an inconsistent checkpoint.

The classic consistent checkpoint algorithm was proposed by Chandy and Lamport and is called the *snapshot algorithm* (12). In the snapshot algorithm, to initiate a checkpoint, a marker message is sent to any one process. When a process gets a marker message for the first time, it checkpoints itself and then sends out marker messages to all the processes it communicates with. If a process receives a marker message subsequent to its first time, it ignores the message. It can be shown that the markers eventually disappear, and when the markers disappear, all processes have recorded a set of consistent checkpoints. Of course many other checkpointing algorithms have been propose since then, having characteristics and features greater that the basic algorithm outlined.

Distributed Deadlocks

Resource management in operating systems can lead to deadlocks. A resource is any entity, such as files, peripherals, memory, and so on. Deadlocks occur for instance when processes acquire locks on resources. For example, suppose a process P_1 locks resource x and then process P_2 locks resource y. Thereafter, process P_1 requests a lock on x and process P_2 requests a lock on y. Neither P_1 nor P_2 can progress any further and has to wait forever. This situation is called a *deadlock,* and it needs to be detected and then resolved by terminating one of the processes. Deadlock detection on centralized systems is easier than deadlock detection on distributed systems.

Consider the following situation, similar to the deadlock described previously, but in the context of a distributed system. A process P_1 requests and obtains a lock on resource x. The resource x is located on a machine Mx and hence is controlled by a lock server running on machine Mx. Now, process P_2 requests and obtains a lock on resource y, which is located on a machine My and controlled by a lock server on machine My. Then process P_1 requests a lock on y and process P_2 requests a lock on x.

This situation is a deadlock. However, the lock servers cannot detect this deadlock by themselves. At the lock server on machine Mx, a process (P_1) holds a lock on x and another process (P_2) has requested a lock on x. This is a perfectly normal, legal situation that is not a deadlock. Similarly, there is no deadlock at machine My. However, a global or distributed deadlock exists, involving two lock servers.

In a system consisting of a large number of lock servers and large numbers of processes and resources, detection of deadlocks becomes a serious issue. Most early distributed deadlock detection algorithms tried to consolidate the data about resource allocation from multiple lock servers in order to find deadlocks. Such algorithms proved to be complicated, expensive in terms of computational complexity, and prone to detect deadlocks even if there are no deadlocks (a phenomenon called *false deadlocks*).

A distributed deadlock detection algorithm by Chandy and Misra was a breakthrough that solved the deadlock problem in a simple fashion. The solution is called the *probe algorithm* (13). In this scheme, a process waiting for a resource sends a probe message to the lock server handling the resource. The lock server forwards the probe to a process that is currently holding the resource. When a process receives a probe, and the process is not currently waiting for a resource, it ignores the probe. If the process is currently waiting for a resource, then it forwards the probe to the lock server that controls the resource. If the originator of the probe gets the probe returned to it, then there is a deadlock. A careful implementation of this protocol can be shown to be free from detection of false deadlocks.

Distributed Concurrency Control

Concurrency control is a mechanism by which the integrity of data is preserved in spite of concurrent access by multiple processes. Concurrently, control is necessary in both single computer systems and distributed systems. In distributed system, the issues are somewhat more complicated as the data may be stored at many different sites.

Concurrency control ensures serializability. Serializability is a property that ensures that the concurrent execution of a set of processes has results that are equivalent to some serial execution of the same set of processes. Serializability is an important property for any system that handles persistent, interrelated data. Provision of serializability is made possible by many techniques, the two most well known are two-phase locking and timestamping.

In the two phase commit scheme, a process that reads or writes data has to obtain a lock on the data item it accesses before it can access the data item, and may release the lock after the access is over. If multiple data items are accessed, then no lock can be released until all locks have been acquired. This ensures serializable updates to the data.

In the timestamp scheme, all data items bear two timestamps, the read-timestamp and the write-timestamp. All processes or transactions also bear timestamps. The process timestamp is the time at which the process was created. The read-timestamp on a data item is the value, which is the largest of all the process timestamps, of processes which have read the data item. The write-timestamp is equal to the process timestamp of the process that last wrote this data item.

The timestamp protocol works as follows. Suppose a process bearing a timestamp pt wants to read a data time with a read-timestamp rt and a write timestamp wt. If $pt < wt$ then the process is aborted or restarted. Otherwise it is allowed to read the item, and if $pt > rt$ then the read timestamp of the item is updated to be equal to rt. If the process tried to write a new value to the data item, then pt must be higher than both rt and wt (else the process is aborted). After the write, both read and write timestamps of the data item are set to pt. The timestamp protocol is termed an *optimistic protocol,* as it does not have any locking delays and all operations are processed immediately or aborted.

The two-phase locking and timestamp protocol can be adapted to distributed systems. To implement two-phase locking, one or more distributed lock servers have to be provided. If multiple lock servers are provided, then distributed deadlock detection has to be added. In addition, the two-phase commit protocol may have to be used for consensus (next section).

To make timestamping work in a distributed system, there needs to be a mechanism to provide system-wide unique timestamps. This is of course possible by using vector clocks as the timestamp. Even Lamport clocks can be used, but to ensure uniqueness, the site identifier of the site that assigns the timestamp is appended to the end of the timestamp.

Distributed Consensus

Consensus is a problem unique to distributed systems. The reason is that distributed systems are composed of separate autonomous systems that need to cooperate. At the times they need to cooperate, there is often a need to agree on something. Suppose there is a file containing the value 0 (zero) on three machines. A process wants to update the value to 1 on all three machines. It tells servers on all the three machines to perform the update. The servers now want to ensure all of them updates, or none of them does it (to preserve consistency). So they need to agree (or arrive at a consensus) to either perform the operation (flip the 0 to 1) or abort the operation (leave it as 0).

In theory, it can be shown that consensus in distributed system is impossible to achieve if there is any chance of loosing messages on the network. The proof is quite involved, but consider the following conversation:

Machine 1 to Machine 2: Flip the bit from 0 to 1, and tell me when you are done so that I will flip it too.

Machine 2 to Machine 1: OK, I have flipped it. But, please acknowledge this message, or else I will think you did not get my reply and you chose not to flip—in which case I will flip mine back to 0.

Machine 1 to Machine 2: Everything is fine. Got your message. But, please acknowledge this message, as I need to know that you got this message, or you may flip the bit back.

Machine 2 to Machine 1: Got it. But now I need another acknowledgment, to ensure . . .

As is obvious, this bickering continues forever. It can be shown that there is no finite length sequence of messages that achieves consensus, even if messages are not lost, as long as there is a *fear* of a message getting lost.

In reality, however, there is need for consensus, and impossibility is not a deterrence. Many systems just assume messages are not lost and thus implement consensus trivially (Machine 1 tells Machine 2 to flip it and assumes it will be done). In more critical applications, the two-phase commit protocol is used.

The two-phase commit protocol works as follows. A machine is selected as the leader (e.g., the one that started the process, that made updates) and the rest of the machines are cohorts. That leader tells all the cohorts to "flip the bit". All of them flip it, and retain a copy of the old value and send an OK to the coordinator. This is called the *pre-commit phase.* At this point, all the cohorts have the old value and the new value. After all the OKs are received, the leader sends a *commit* message which causes all the cohorts to install the new (flipped) value. If some OKs are not received, the leader tells all the cohorts to abort, that is install the old value back. It can be shown that this protocol (with some extensions for failure handling) works for most cases of message loss and machine failure.

Replication Control

In distributed systems, data are often replicated; that is, multiple copies of the same data are stored on multiple sites. This is for reliability, performance, or both. Performance is enhanced if regularly accessed data are scattered over the network, rather than in one place—it evens out the access load. In addition, if one site having the data fails then the date is still available from the other sites. Replication works very well for read-only data. But, to be useful, replication should work with read-write data also. Replication control protocols ensure that data replication is consistent, in spite of failures for read-write data. There are many protocols, a few are outlined below.

Read One, Write All. In this scheme, a reader can read from any copy, but a writer has to update all copies. If not all copies are available, the writer cannot update. Most commonly used.

Primary Copy. A variant of the previous, read any copy, write to the primary copy. The machine holding the primary copy then propagates the update.

Read Majority Write Majority. If there are N copies, then read $N/2 + 1$ copies and take the value from the most recent of the copies. Writing to any of the $N/2 + 1$ copies is good enough.

Voting. Each copy has a certain number of votes. The total number of votes is v. Choose a read quorum r and a write quorum w such that $r + w = q + 1$. Now, to access, find enough copies such that the total vote is equal (or greater) than r for reading, and w for writing.

Depending on the read traffic, the write traffic, and the failure probabilities, one of these protocols is chosen. Note that voting is a general protocol, where setting the votes of each item to 1 and r to 1 and w to N makes it the read-one write-all protocol. Similarly, it can mimic the majority protocol. There are other protocols that are more general than voting (such as quorum consensus).

SYSTEM FEATURES

The following paragraphs outline the salient features of a set of network (or distributed) operating systems that either are in operation or have significant contributions to the state of the art.

Amoeba

Amoeba, developed at Vrije University (6), is an operating system using a microkernel design, supporting very fast message passing designed to utilize processor farms. A processor farm is a set of rack-mounted single-board computers connected by regular networking (Ethernet). Amoeba makes the collection machines look like one fast timesharing system. It also provides support for threads, RPC, group communication, and all other facilities needed for networking. Amoeba supports a parallel programming language called Orca.

Clouds

Clouds, developed at Georgia Tech (14), is a system designed to support persistent objects that are large grained. Each object is an address space that is backed up on disk and hence is persistent. The system paradigm uses a thread-object model, where threads are distributed and can access objects via a modified RPC mechanism. The object invocation causes the thread to move between address spaces rather than use a server for processing the RPC request. The entire system is supported on top of a low-level distributed shared memory mechanism thus making all objects available at all computers. Services are built into objects and can be accessed using the RPC mechanism. Message passing is not supported at the API level. Clouds has been used for research in reliability, transaction processing, replication, and distributed debugging.

Mach

Mach, developed at Carnegie-Mellon University (7), is a Unix compatible operating system that is built on a microkernel. The microkernel supports message passing, tasks, and threads. Mach supports an innovative user-level external paging system that causes messages to be sent to a paging process whenever there is a page-fault generated by a user process. These external pagers allowed Mach to support a variety of emulation features. The Unix operating system is supported on top of Mach as a user-level process, providing the Unix service. Mach is also heavily customizable, making it an ideal platform for research with operating systems.

Sprite

Sprite, developed at University of California, Berkeley (15), is an operating system that provides a single system image to a cluster of workstations. Much of the focus of research with Sprite has been directed at improving file system performance. As a result, Sprite provides a very high performance file system through client and server caching. It has process migration to take advantage of idle machines. It was used as a testbed for research in log-structured file systems, striped file systems, crash recovery, and RAID file systems.

Unix

Unix is a commercial product of Unix Systems Laboratories. Various other companies sell variants of Unix, using other trade names, the most well-known being SunOS/Solaris. SunOS was the first system to provide a commercial, robust, full-featured network file system (NFS). Linux is a free Unix compatible operating system. The kernel of Unix is monolithic and most network-based services are added as separate user processes. Unix is an older operating system, adapted for network use. Because of the prevalence of Unix in research institutions, all services developed for networking are developed on Unix platforms first. Hence, everything is available for Unix, though not from the commercial providers of Unix. Unix is the mainstay of network operating systems in the academic and research communities.

V-System

The V-System, developed at Stanford University (8), is a microkernel operating system with support for fast message passing. Services are added to V by running user-level servers. The innovative use of low-latency protocols for intermachine messaging provides V with excellent performance on a networked environment. Also innovative is the uniform support for input–output, a capability based naming scheme, and the clean design of the kernel.

Windows NT

Windows NT is a commercial product of Microsoft Corporation. This operating system has a hybrid kernel; that is, the inner core of the operating system follows the microkernel technology, but the services are not at the user-level. Services are added to Windows NT as modules called DLLs (dynamic link libraries). The operating system is extensible and allows for a variety of pluggable modules at the level of device drivers, kernel extensions, as well as services at the user level. Windows NT provides many of the services described in this article in a commercial product and competes with the various forms of Unix in the marketplace. Windows NT also has the ability of running applications written for DOS, Windows 3.1, and Windows 95, all of which are completely different operating systems. For network use, Windows NT provides file service, name service, replication service, RPC service, and messaging using several protocols.

RELATED TOPICS

Distributed Operating Systems

Distributed operating systems are network operating systems with significantly more integration between the autonomous operating system running on each machine. The distributed operating system is hence able to provide services that are beyond the capability of network operating systems. A few of the additional facilities are summarized.

Dynamic Distributed Data Placement. A data item of file is located close to where it is used. Its location changes dynamically as its usage pattern changes. The logical location (such as a file is in one particular directory) is not an indicator of its physical locations. For example, a directory may contain three files, but the files may be located at three different machines, at some point in time.

Process Scheduling. When a process is started, it is not started on the same machine as its parent, but the process scheduler decides where to start the process. The chosen machine may be a machine with the lightest load, or a machine that is close to the data the process will be accessing.

Process Migration. Processes may move from machine to machine (automatically) depending on its data access patterns, or resource needs, or just for load balancing.

Fault Tolerance. Failures of sites do not affect any of the computations. Failed computations are automatically restarted, inaccessible data are made available though replicated copies. Users connected to the failed machine are transparently relocated.

Distributed Parallel Processing Systems

The bastion of parallel processing used to be large, expensive machines called *parallel processors.* The advent of network operating systems has shifted the focus of parallel processing platforms to cheaper hardware—a network of smaller machines. Parallel processing involves splitting a large task into smaller units, each of which can be executed on a separate processor, concurrently. This method uses more hardware, but causes the task to run faster and complete quicker. Parallel processing is very necessary in applications such as weather forecasting, space exploration, image processing, large database handling, and many scientific computations.

Parallel processing on network operating system uses toolkits, also known as middleware, which sit between the application and the operating system and manage the control flow and the data flow. A particularly popular package is called PVM (parallel virtual machine) (16). PVM augments the message passing system provided by the operating system with simpler to use primitives that allow control of spawning processes on remote machines, transmission of data to the machine, and collection of results of the computations. Another package with similar characteristics is MPI (17). An interesting system that uses a radically different approach to parallel processing is Linda (18). Linda integrates the notion of work and data into a unified concept called the tuple-space. The tuple-space contains work tuples and data tuples. Processes called *workers* run on many machines and access the tuple-space to get work, to get input, and to store the results.

Some recent parallel processing system use distributed shared memory to hold the data, mimicking the facilities available on the large parallel processors. Such systems are easier to program as they insulate the programmer from the idiosyncrasies of data placement and data transmission. TreadMarks (19) is a product that provides a high-performance distributed shared memory system using a method called release consistency. Calypso (20) is another system that supports easy to program parallel processing, and also provides load balancing and fault tolerance with no additional cost. Calypso uses a manager–worker model that creates a logical parallel processor, and can dynamically change the number of workers depending on physical network characteristics. Other systems that are in use include Amber, Avalanche, GLU, P4, Piranha, and Quarks.

BIBLIOGRAPHY

1. A. S. Tannenbaum, *Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1995.
2. P. K. Sinha, *Distributed Operating Systems: Concepts and Design.* New York: IEEE Press, 1997.
3. E. Levy and A. Silberschatz, Distributed file systems: Concepts and examples, *ACM Comput. Surv.,* **22** (4): 321–374, 1990.
4. K. Birman, The process group approach to reliable distributed computing, *Commun. ACM,* **36** (12): 37–53, 1993.
5. K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Trans. Comput. Syst.,* **7** (4): 321–359, 1989.
6. A. S. Tanenbaum et al., Experience with the amoeba distributed operating system, *Commun. ACM,* **33** (12): 46–63.
7. M. Accetta et al., Mach: A new kernel foundation for Unix development, *Proc. Summer Usenix Conf.,* 1990.
8. D. R. Cheriton, The V Distributed System, *Commun. ACM,* **31** (3): 314–333, 1988.
9. N. Lynch, *Distributed Algorithms,* San Francisco: Morgan Kaufman Publishers, 1997.
10. L. Lamport, Time, clocks and ordering of events in a distributed system, *Commun. ACM,* **21** (7): 558–565, 1978.
11. M. Maekawa, A. E. Oldehoft, and R. R. Oldehoft, *Operating Systems: Advanced Concepts,* Menlo Park, CA: Benjamin-Cummings, 1987.
12. K. M. Chandy and L. Lamport, Distributed snapshots, *ACM Trans. Comp. Sys.,* **3** (1): 63–75, 1985.
13. K. M. Chandy, J. Misra, and L. M. Haas, Distributed deadlock detection, *ACM Trans. Comp. Sys.,* **1** (2): 144–156, 1983.
14. P. Dasgupta et al., *The Clouds Distributed Operating System,* IEEE Computer, November 1991.
15. M. Nelson, B. Welch, and J. Ousterhout, Caching in the Sprite network file system, *ACM Trans. Comp. Syst.,* **6** (1): 134–154, 1988.
16. A. Geist et al., *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing,* Cambridge, MA: MIT Press, 1994.
17. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* MIT Press, 1994.
18. D. Gelernter, Programming for advanced computing, *Sci. Amer.,* **257** (4): 65–71, 1987.

19. C. Amza et al., TreadMarks: Shared memory computing on networks of workstations, *IEEE Comp.,* 29 (2): 18–28, 1996.

20. A. Baratloo, P. Dasgupta, and Z. M. Kedem, Calypso: A novel software architecture for high performance parallel processing on workstation networks, *4th Int. Conf. High Performance Distributed Comput.,* 1995.

PARTHA DASGUPTA
Arizona State University