Survey on Distributed Operating Systems: A Real Time Approach

Shailesh Khapre, Rayer Jean, J. Amudhavel, D. Chandramohan, P. Sujatha and V. Narasimhulu

Department of Computer Science, Pondicherry Central University,
Pondicherry - 605014, India.
{shaileshkhaprerkl, jeanrayar, amudhavel86, chandrumeister, spothula, narasimhavasi}@gmail.com

***Abstract*:** Today's typical computing environment has changed from mainframe systems to small computing systems that often cooperate via communication networks. Distributed Operating Systems Concepts and Design addresses the organization and principles of distributed computing systems. Although it does not concentrate on any particular operating system or hardware, it introduces the major concepts of distributed operating systems without requiring that readers know all the theoretical or mathematical fundamentals. Distributed operating systems have many aspects in common with centralized ones, but they also differ in certain ways. This paper is intended as an introduction to distributed operating systems, and especially to current university research about them. After a discussion of what constitutes a distributed operating system and how it is distinguished from a computer network, various key design issues are discussed.

***Keywords*:** Distributed Systems, Modern Operating Systems.

1. Introduction

Everyone agrees that distributed systems are going to be very important in the future. Unfortunately, not everyone agrees on what they mean by the term "distributed system." In this paper we present a view point widely held within academia about what is and is not a distributed system, we discuss numerous interesting design issues concerning them, and finally we conclude with a fairly close look at some experimental distributed systems that are the subject of ongoing research at universities[1]. A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs). The key concept here is transparency. In other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a "virtual uniprocessor," not as a collection of distinct machines.

To make the contrast with distributed operating systems stronger, let us briefly look at another kind of system, which we call a "network operating system." A typical configuration for a network operating system would be a collection of personal computers along with a common printer server and file server [35] for archival storage, all tied together by a local network. Users are typically aware of where each of their files are kept and must move files between machines with explicit "file transfer" commands, instead of having file placement managed by the operating system. The system has little or no fault tolerance [3][6][17]; if 1 percent of the personal computers crashes, 1 percent of the users are out of business, instead of everyone simply being able to continue normal work, albeit with 1 percent worse performance.

1.1 Goals and Problem

A more fundamental problem in distributed systems is the lack of global state information. It is generally a bad idea to even try to collect complete information about any aspect of the system in one table. Lack of up-to-date information makes many things much harder. It is hard to schedule the processors optimally if you are not sure how many are up at the moment. Many people, however, think that these obstacles can be overcome in time, so there is great interest in doing research on the subject.

2. Network Operating System

Before starting our discussion of distributed operating systems, it is worth first taking a brief look at some of the ideas involved in network operating systems, since they can be regarded as primitive forerunners. Although attempts to connect computers together have been around for decades, networking really came into the limelight with the ARPANET in the early 1970s. The original design did not provide for much in the way of a network operating system. Instead, the emphasis was on using the network as a glorified telephone line to allow remote login and file transfer. Later, several attempts were made to create net- work operating systems, but they never were widely used. In more recent years, several research organizations have connected collections of minicomputers running the UNIX operating system into a network operating system, usually via a local network [9] [19] [29] gives a good survey of these systems, which we shall draw upon for the remainder of this section. As we said earlier, the key issue that distinguishes a network operating system from a distributed one is how aware the users are of the fact that multiple machines are being used. This visibility occurs in three primary areas: the file system, protection, and program execution. Of course, it is possible to have systems that are highly transparent in one area and not at all in the other, which leads to a hybrid form.

2.1 File System

When connecting two or more distinct systems together, the first issue that must be faced is how to merge the file systems [18] [19] [22] [37]. Three approaches have been tried.

The first approach is not to merge them at all. Going this route means that a program on ma- chine A cannot access files on machine B by making system calls. Instead, the user must run a special file transfer program that copies the needed remote files to the local machine, where they can then be accessed normally. Sometimes remote printing and mail is also handled this way. One of the best-known examples of networks that primarily supports file transfer [11] and mail via special programs, and not system call access to remote files, is the UNIX "uucp" program, and its network, USENET.

The next step upward in the direction of a distributed file system is to have adjoining file systems. In this approach, programs on one machine can open files on another ma- chine by providing a path name telling where the file is located. For example, one could say open(' '/machinel/pathname' ', READ); open("machine/pathname", READ); open('f/. ./machinel/pathname", READ); The latter naming scheme is used in the Newcastle Connection [19] and is derived from the creation of a virtual "super directory" above the root directories of all the connected machines. Thus "/. ." means start at the local root directory and go upward one level (to the super directory), and then down to the root directory of machine." In Figure 1, the root directory of three machines, A, B, and C are shown, with a super directory above them. To access file x from machine C, one could say open (' '/. ./C/x' ', READ-ONLY). In the Newcastle system, the naming tree is actually more general, since "machine 1" may really be any directory, so one can attach a machine as a leaf anywhere in the hierarchy, not just at the top.

The third approach is the way it is done in distributed operating systems, namely, to have a single global file system visible from all machines. When this method is used, there is one "bin" directory for binary programs, one password file, and so on. When a program wants to read the pass- word file it does something like open (' '/etc/passwd' ', READ-ONLY) without reference to where the file is. It is up to the operating system to locate the file and arrange for transport of data as they are needed. LOCUS is an example of a system using this approach. The convenience of having a single global name space is obvious. In addition, this approach means that the operating system is free to move files around among machines to keep all the disks equally full and busy, and that the system can maintain.

Replicated copies of files if it so chooses. When the user or program must specify the machine name, the system cannot decide on its own to move a file to a new machine because that would change the (user visible) name used to access the file. Thus in network operating system, control over file placement must be done manually by the users, whereas in a distributed operating system it can be done automatically by the system itself.

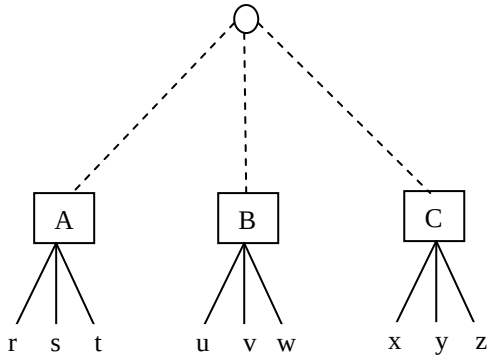

Figure 1. A (virtual) superdirectory above the root directory provides access to remote files.

2.2 Protection

Closely related to the transparency of the file system is the issue of protection. UNIX and many other operating systems assign a unique internal identifier to each user. Each file in the file system has a little table associated with it (called an i-node in UNIX) telling who the owner is, where the disk blocks are located, etc. If two previously independent machines are now connected, it may turn out that some internal User Identifier (UID), for example, number 12, has been assigned to a different user on each machine. Consequently, when user 12 tries to access a remote file, the remote file system cannot see whether the access is permitted since two different users have the same UID. One solution to this problem is to require all remote users wanting to access files on machine X to first log onto X using a user name that is local to X. When used this way, the network is just being used as a fancy switch to allow users at any terminal to log onto any computer, just as a telephone company switching center allows A better approach is to have the operating system provide a mapping between UIDs, so that when a user with UID 12 on his or her home machine accesses a remote machine on which his or her UID is 15, the remote machine treats all accesses as though they were done by user 15. This approach implies that sufficient tables are provided to map each user from his or her home (machine, UID) pair to the appropriate UID for any other machine (and that messages cannot be tampered with).any subscriber to call any other subscriber. This solution is usually inconvenient for people and impractical for programs, so something better is needed. The next step up is to allow any user to access files on any machine without having to log in, but to have the remote user appear to have the UID corresponding to "GUEST" or "DEMO" or some other publicly known login name. Generally such names have little authority and can only access files that have been designated as readable or writable by all users. In a true distributed system there should be a unique UID for every user, and that UID should be valid on all machines with- out any mapping. In this way no protection problems arise on remote accesses to files; as far as protection goes, a remote access can be treated like a local access with the same UID. The protection issue makes the difference between a network operating system and a distributed one clear: In one case there are various machines, each with its own user-to-UID mapping and in the other there is a single, system wide mapping that is valid everywhere.

2.3 Execution Location

Program execution is the third area in which machine boundaries are visible in network operating systems. When a user or a running program wants to create a new process, where is the process created? At least four schemes have been used thus far. The first of these is that the user simply says "CREATE PROCESS" in one way or another, and specifies nothing about where. Depending on the implementation, this can be the best or the worst way to do it. In the most distributed case, the system chooses a CPU by looking at the load, location of files to be used, etc. In the least distributed case, the system always runs the process on one specific machine (usually the machine on which the user is logged in). The second approach to process location is to allow users to run jobs on any machine by first logging in there. In this model, processes on different machines cannot communicate or exchange data, but a simple manual load balancing is possible. The third approach is a special command that the user types at a terminal to cause a program to be executed on a specific ma- chine. A typical command might be remote vax4 who to run program on machine vax4. In this arrangement, the environment of the new process is the remote machine. In other words, if that process tries to read or write files from its current working directory, it will discover that its working directory is on the remote machine, and that files that were in the parent process's directory are no longer present. Similarly, files written in the working directory will appear on the remote machine, not the local one. The fourth approach is to provide the "CREATE PROCESS" system call with a parameter specifying where to run the new process, possibly with a new system call for specifying the default site. As with the previous method, the environment will generally be the remote machine. In many cases, signals and other forms of interprocess communication between processes do not work properly among processes on different machines. A final point about the difference between network and distributed operating systems is how they are implemented. A common way to realize a network operating system is to put a layer of software on top of the native operating systems of the individual machines. For example, one could write a special library package that would intercept all the system calls and decide whether each one was local or remote [19] Although most system calls can be handled this way without modifying the kernel, invariably there are a few things, such as interprocess signals, interrupt characters (e.g., BREAK) from the keyboard, etc., that are hard to get right. In a true distributed operating system one would normally write the kernel from scratch. 1.4 An Example: The Sun Network File System To provide a contrast with the true distributed systems described later in this paper, in this section we look briefly at a network operating system that runs on the Sun Microsystems' workstations. These work stations are intended for use as personal computers. Each one has a 68000 series CPU, local memory, and a large bit- mapped display. Workstations can be configured with or without local disk, as desired. All the workstations run a version of 4.2BSD UNIX specially modified for networking. This arrangement is a classic example of a network operating system: Each computer runs a traditional operating system, UNIX, and each has its own user(s), but with extra features added to make networking more convenient. During its evolution the Sun system has gone through three distinct versions, which we now describe. In the first version each of the work- stations was completely independent from all the others, except that a program rep was provided to copy files from one work- station to another. By typing a command such as rep Ml:/usr/jim/file.c M2:/usr/ast/f.c it was possible to transfer whole files from one machine to another. In the second version, Network Disk (ND), a network disk server was provided to support diskless workstations. Disk space on the disk server's machine was divided into disjoint partitions, with each partition acting as the virtual disk for some (diskless) workstation. Whenever a diskless workstation needed to read a file, the request was processed locally until it not down to the level of the device driver, it which point the block needed was retrieved by sending a message to the remote disk server. In effect, the network was merely being used to simulate a disk controller. With this network disk system, sharing of disk partitions was not possible. The third version, the Network File Sys- tem (NFS), allows remote directories to be mounted in the local file tree on any work- station. By mounting, say, a remote directory "dot" on the empty local directory "/usr/doc," all subsequent references to "/usr/doc" are automatically routed to the remote system. Sharing is allowed in NFS, so several users can read files on a remote machine at the same time. To prevent users from reading other people's private files, a directory can only be mounted remotely if it is explicitly exported by the workstation it is located on. A directory is exported by entering a line for it in a file "/etc/exports." To improve performance of remote access, both the client ma- chine and server machine do block caching. Remote services can be located using a Yellow Pages server that maps service names onto their network locations. The NFS is implemented by splitting the operating system up into three layers. The top layer handles directories, and maps each path name onto a generalized i-node called a unode consisting of a (machine, i-node) pair, making each vnode globally unique.

Vnode numbers are presented to the middle layer, the virtual file system (VFS). This layer checks to see if a requested vnode is local or not. If it is local, it calls the local disk driver or, in the case of an ND partition, sends a message to the remote disk server. If it is remote, the VFS calls the bottom layer with a request to process it remotely. The bottom layer accepts requests for accesses to remote vnodes and sends them over the network to the bottom layer on the serving machine. From there they prop- agate upward through the VFS layer to the top layer, where they are reinjected into the VFS layer. The VFS layer sees a request for a local vnode and processes it normally, without realizing that the top layer is actually working on behalf of a remote kernel. The reply retraces the same path in the other direction. The protocol between workstations has been carefully designed to be robust in the face of network and server crashes. Each request completely identifies the file (by its vnode), the position in the file, and the byte count. Between requests, the server does not maintain any state information about which files are open or where the current file position is. Thus, if a server crashes and is rebooted, there is no state information that will be lost. The ND and NFS facilities are quite different and can both be used on the same workstation without conflict. ND works at a low level

and just handles remote block I/O without regard to the structure of the information on the disk. NFS works at a much higher level and effectively takes re- quests appearing at the top of the operating system on the client machine and gets them over to the top of the operating system on the server machine, where they are processed in the same way as local requests.

3. Design Issues

Now we turn from traditional computer systems with some networking facilities added on to systems designed with the intention of being distributed. In this section we look at five issues that distributed systems' designers are faced with: communication primitives, naming and protection, resource management, fault tolerance [4][5][6], services to provide. Although no list could possibly be exhaustive at this early stage of development, these topics should provide a reasonable impression of the areas in which current research is proceeding.

3.1 Communication Primitives

The computers forming a distributed sys- tem normally do not share primary memory, and so communication via shared memory techniques such as semaphores and monitors is generally not applicable. Instead, message passing in one form or another is used [23]. One widely discussed framework for message-passing systems is the IS0 OS1 reference model, which has seven layers, each performing a well- defined function. The seven layers are the physical layer, data- link layer, network layer, transport layer, session layer, presentation layer, and application layer. By using this model it is possible to connect computers with widely different operating systems, character codes, and ways of viewing the world. Unfortunately, the overhead created by all these layers is substantial. In a distributed system consisting primarily of huge mainframes from different manufacturers, connected by slow leased lines (say, 56 kilobytes per second), the overhead might be tolerable. Plenty of computing capacity would be available for running complex protocols, and the narrow bandwidth means that close coupling between the systems would be impossible anyway. On the other hand, in a distributed system consisting of identical microcomputers connected by a lo- megabyte-per second or faster local network, the price of the IS0 model is generally too high. Nearly all the experimental distributed systems discussed in the literature thus far have opted for a different, much simpler model, so we do not mention the IS0 model further in this paper.

3.2 Message Passing

The model that is favored by researchers in this area is the client-server model, in which a client process wanting some service (e.g., reading some data from a tile) ends a message to the server and then waits for a reply message, as shown in Figure 2. In the most naked form the system just pro- vides two primitives: SEND and RE- CEIVE. The SEND primitive specifies the destination and provides a message; the RECEIVE primitive tells from whom a message is desired (including "anyone") and provides a buffer where the incoming message is to be stored. No initial setup is required, and no connection is established, hence no tear down is required.

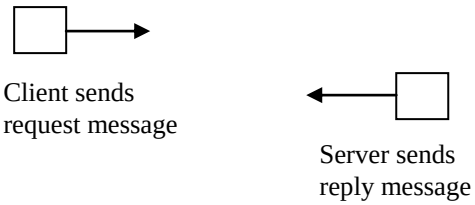

Figure 2. Client-server model of communication

Precisely what semantics these primitives ought to have has been a subject of much controversy among researchers. Two of the fundamental decisions that must be made are unreliable versus reliable and no blocking versus blocking primitives. At one extreme, SEND can put a message out onto the network and wish it good luck. No guarantee of delivery is pro- vided, and no automatic retransmission is attempted by the system if the message is lost. At the other extreme, SEND can handle lost messages, retransmissions, and acknowledgments internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

Blocking versus Non blocking Primitives: The other choice is between no blocking and blocking primitives. With nonblocking primitives, SEND returns control to the user program as soon as the message has been queued for subsequent transmission (Or a copy made). If no copy is made, any Changes the program makes to the data before or (heaven forbid) while they are being sent are made at the program's peril. When the message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused. The corresponding RECEIVE primitive signals a willingness to receive a message and provides a buffer for it to be put into. When a message has arrived, the program is informed by interrupt, or it can poll for status continuously or go to sleep until the interrupt arrives. The advantage of these non blocking primitives is that they provide the maximum flexibility: Programs can compute and perform message I/O in parallel in any way they want. Non blocking primitives also have a disadvantage: They make programming tricky and difficult. Irreproducible, timing-dependent programs are painful to write and awful to debug. Consequently, many people advocate sacrificing some flexibility and efficiency by using blocking primitives. A. blocking SEND does not return control to the user until the message has been sent (unreliable blocking primitive) or until the message has been sent and an acknowledgment received (reliable blocking primitive). Either way, the program may immediately modify the buffer without danger. A blocking RECEIVE does not return control until a message has been placed in the buffer. Reliable and unreliable RECEIVES differ in that the former automatically acknowledges receipt of a message, whereas the latter does not. It is not reasonable to combine a reliable SEND with an unreliable RECEIVE, or vice versa; so the system designers must make a choice and provide one set or the other. Blocking and non- blocking primitives do not conflict, so

there is no harm done if the sender uses one and the receiver the other. Receiver, although buffered message passing can be implemented in many ways, a typical approach is to provide users with a system call CREATEBUF, which creates a kernel buffer, sometimes called a mailbox, of a user-specified size. To communicate, a sender can now send messages to the receiver's mailbox, where they will be buffered until requested by the receiver. Buffering is not only more complex (creating, destroying, and generally managing the mailboxes), but also raises issues of protection, the need for special high-priority interrupt messages, what to do with mail- boxes owned by processes that have been killed or died of natural causes, and more. A more structured form of communication is achieved by distinguishing requests from replies. With this approach, one typically has three primitives: SEND-GET, GET-REQUEST, and SEND-REPLY. SEND-GET is used by clients to send re- quests and get replies. It combines a SEND to a server with a RECEIVE to get the server's reply. GET-REQUEST is done by servers to acquire messages containing work for them to do. When a server has carried the work out, it sends a reply with SEND-REPLY. By thus restricting the message traffic and using reliable, blocking primitives, one can create some order in the chaos.

3.3 Remote Procedure Call (RPC)

The next step forward in message-passing systems is the realization that the model of "client sends request and blocks until server sends reply" looks very similar to a traditional procedure call from the client to the server. This model has become known in the literature as "remote procedure call" and has been widely discussed [10] [12]. The idea is to make the semantics of inter- machine communication as similar as possible to normal procedure calls because the latter is familiar and well understood, and has proved its worth over the years as a tool for dealing with abstraction. It can be viewed as a refinement of the reliable, blocking SEND-GET, GET-REQUEST, SENDREP primitives, with a more user-friendly syntax. The remote procedure call can be organized as follows. The client (calling program) makes a normal procedure call, say, p(x, y) on its machine, with the intention of invoking the remote procedure p on some other machine. A dummy or stub procedure p must be included in the caller's address space, or at least be dynamically linked to it upon call. This procedure, which may be automatically generated by the compiler, collects the parameters and packs them into a message in a standard format. It then sends the message to the remote machine (using SEND-GET) and blocks, waiting for an answer (see Figure 3). At the remote machine, another stub procedure should be waiting for a message using GET-REQUEST. When a message comes in, the parameters are unpacked by an input-handling procedure, which then makes the local call p(x, y). The remote procedure p is thus called locally, and so its normal assumptions about where to find parameters, the state of the stack, etc., are identical to the case of a purely local call. The only procedures that know that the call is remote are the stubs, which build and send the message on the client side and disassemble and make the call on the server side. The result of the procedure call follows an analogous path in the reverse direction.

Although at first glance the remote procedure call model seems clean and simple, under the surface there are several problems. One problem concerns parameter (and result) passing. In most programming languages, parameters can be passed by value or by reference. Passing value parameters over the network is easy; the stub just copies them into the message and off they go. Passing reference parameters (pointers) over the network is not so easy. One needs a unique, system wide pointer for each object so that it can be remotely accessed. For large objects, such as files, some kind of capability mechanism [36] could be set up, using capabilities as pointers.

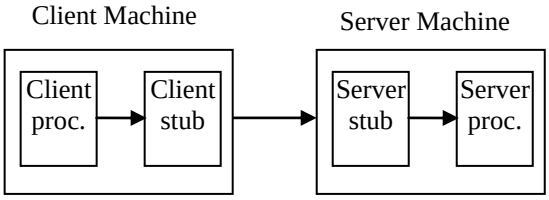

Figure 3. Remote procedure call.

Leans, the amount of overhead and mechanism needed to create a capability and send it in a protected way is so large that this solution is highly undesirable. Still another problem that must be dealt with is how to represent parameters and results in messages. This representation is greatly complicated when different types of machines are involved in a communication. A floating-point number produced on one machine is unlikely to have the same value on a different machine, and even a negative integer will create problems between the l's complement and 2's complement machines. Converting to and from a standard for- mat on every message sent and received is an obvious possibility, but it is expensive and wasteful, especially when the sender and receiver do, in fact, use the same internal format. If the sender uses its internal format (along with an indication of which format it is) and lets the receiver do the conversion, every machine must be pre- pared to convert from every other format. When a new machine type is introduced, much existing software must be upgraded. Any way it is done, with remote procedure call (RPC) or with plain messages, it is an unpleasant business. Some of the unpleasantness can be hid- den from the user if the remote procedure call mechanism is embedded in a programming language with strong typing, so that the receiver at least knows how many parameters to expect and what types they have. In this respect, a weekly typed language such as C, in which procedures with a variable number of parameters are common, is more complicated to deal with. Still another problem with RPC is the issue of client-server binding. Consider, for example, a system with multiple file servers. If a client creates a file on one of the file servers, it is usually desirable that sub-sequent writes to that file go to the file server where the file was created. With mailboxes, arranging for this is straight- forward. The client simply addresses the WRITE messages to the same mailbox that the CREATE message was sent to. Since each file server has its own mailbox, there is no ambiguity. When RPC is used, the situation is more complicated, since the entire client does is put a procedure call such as write (File Descriptor, Buffer Address, Byte Count); in his program. RPC intentionally

hides all the details of locating servers from the client, but sometimes, as in this example, the details are important. In some applications, broadcasting and multicasting (sending to a set of destinations, rather than just one) is useful. For example, when trying to locate a certain person, process, or service, sometimes the only approach is to broadcast an inquiry message and wait for the replies to come back. RPC does not lend itself well to sending messages to sets of processes and getting answers back from some or all of them. The semantics are completely different. Despite all these disadvantages, RPC re- mains an interesting form of communication and much current research is being addressed toward improving it and solving the various problems discussed above.

3.4 Naming and Protection

All operating systems support objects such as files, directories, segments, mailboxes, processes, services, servers, nodes, and I/O devices. When a process wants to access one of these objects, it must present some kind of name to the operating system to specify which object it wants to access. In some instances these names are ASCII strings designed for human use; in others they are binary numbers used only internally. In all cases they have to be managed and protected from misuse.

3.4.1 Naming and Protection

Naming [33] can best be seen as a problem of mapping between two domains. For example, the directory system in UNIX provides a mapping between ASCII path names and i-node numbers. When an OPEN system call is made, the kernel converts the name of the file to be opened into its i-node number. Internal to the kernel, files are nearly always referred to by i-node number, not ASCII string. Just about all operating systems have something similar. In a distributed system a separate name server is sometimes used to map user-chosen names (ASCII strings) onto objects in an analogous way. Another example of naming is the map- ping of virtual addresses onto physical ad- dresses in a virtual memory system. The paging hardware takes a virtual address as input and yields a physical address as out- put for use by the real memory. In some cases naming implies only a single level of mapping, but in other cases it can imply multiple levels. For example, to use some service, a process might first have to map the service name onto the name of a server process that is prepared to offer the service. As a second step, the server would then be mapped onto the number of the CPU on which that process is running. The mapping need not always be unique, for example, if there are multiple processes prepared to offer the same service.

3.4.2 Name Servers

In centralized systems, the problem of naming can be effectively handled in a straight- forward way. The system maintains a table or database providing the necessary name-to-object mappings. The most straightforward generalization of this approach to distributed systems is the single name server model. In this model, a server accepts names in one domain and maps them onto names in another domain. For example, to locate services in some distributed systems, one sends the service name in ASCII to the name server, and it replies with the node number where that service can be found, or with the process name of the server process, or perhaps with the name of a mailbox to which requests for service can be sent. The name server's database is built up by registering services, processes, etc., that want to be publicly known. File directories can be regarded as a special case of name service. Although this model is often acceptable in a small distributed system located at a single site, in a large system it is undesirable to have a single centralized component (the name server) whose demise can bring the whole system to a grinding halt. In addition, if it becomes overloaded, performance will degrade. Furthermore, in a geo- graphically distributed system that may have nodes in different cities or even countries, having a single name server will be inefficient owing to the long delays in accessing it. The next approach is to partition the system into domains, each with its own name server. If the system is composed of multiple local networks connected by gate- ways and bridges, it seems natural to have one name server per local network. One way to organize such a system is to have a global naming tree, with files and other objects having names of the form: /country/city/network/pathname. When such a name is presented to any name server, it can immediately route the request to some name server in the designated country, which then sends it to a name server in the designated city, and so on until it reaches the name server in the network where the object is located, where the mapping can be done. Telephone numbers use such a hierarchy, composed of country code, area code, exchange code (first three digits of telephone number in North America), and sub- scriber line number. Having multiple name servers does not necessarily require having a single, global naming hierarchy. Another way to organize the name servers is to have each one effectively maintain a table of, for example, (ASCII string, pointer) pairs, where the pointer is really a kind of capability for any object or domain in the system. When a name, say a/b/c, is looked up by the local name server, it may well yield a pointer to another domain (name server), to which the rest of the name, b/c, is sent for further processing (see Figure 4). This facility can be used to provide links (in the UNIX sense) to files or objects whose precise whereabouts is managed by a remote name server. Thus if a file foobar is located in another local network, n, with name server n.s, one can make an entry in the local name server's table for the pair (x, n.s) and then access xlfoobar as though it were a local object. Any appropriately authorized user or process knowing the name xlfoobar could make its own synonym s and then perform accesses using s/x/foobar. Each name server parsing a name that involves multiple name servers just strips off the first component and passes the rest of the name to the name server found by looking up the first component locally. A more extreme way of distributing the name server is to have each machine man-age its own names. To look up a name, one broadcasts it on the network. At each ma- chine, the incoming request is passed to the local name server, which replies only if it finds a match. Although broadcasting is easiest over a local network such as a ring net or CSMA net (e.g., Ethernet), it is also possible over store-and-forward packet switching networks such as the ARPANET [34]. Although the normal use of a name server is to map an ASCII string onto a binary

number used internally to the system, such as a process identifier or machine number, once in a while the inverse mapping is also useful. For example, if a machine crashes, upon rebooting it could present its (hard- wired) node number to the name server to ask what it was doing before the crash, that is, ask for the ASCII string corresponding to the service that it is supposed to be offering so that it can figure out what pro- gram to reboot.

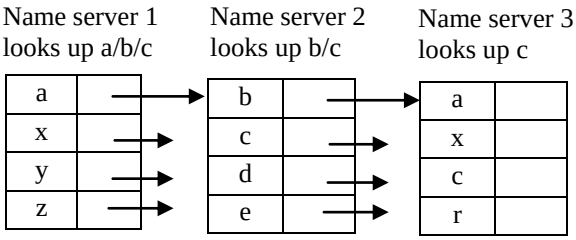

Figure 4. Distributing the lookup of a/b/c over three name servers

3.4.3 Process Allocation

One of the key resources to be managed in a distributed system is the set of available processors. One approach that has been proposed for keeping tabs on a collection of processors is to organize them in a logical hierarchy independent of the physical structure of the network, as in MICROS. This approach organizes the machines like people in corporate, military, academic, and other real-world hierarchies. Some of the machines are workers and others are managers. For each group of k workers, one manager machine (the "department head") is assigned the task of keeping track of who is busy and who is idle. If the system is large, there will be an unwieldy number of department heads; so some machines will function as "deans," riding herd on k department heads. If there are many deans, they too can be organized hierarchically, with a "big cheese" keeping tabs on k deans. This hierarchy can be extended ad infinitum, with the number of levels needed growing logarithmically with the number of workers. Since each processor need only maintain communication with one superior and k subordinates, the information stream is manageable [15]. An obvious question is, "What happens when a department head, or worse yet, a big cheese, stops functioning (crashes)?" One answer is to promote one of the direct subordinates of the faulty manager to fill in for the boss. The choice of which one can either be made by the subordinates themselves, by the deceased's peers, or in a more autocratic system, by the sick man- ager's boss. To avoid having a single (vulnerable) manager at the top of the tree, one can truncate the tree at the top and have a committee as the ultimate authority. When a member of the ruling committee malfunctions, the remaining members promote someone one level down as a replacement. Although this scheme is not completely distributed, it is feasible and works well in practice. In particular, the system is self- repairing, and can survive occasional crashes of both workers and managers without any long-term effects. In MICROS, the processors are mono- programmed, so if a job requiring S processes suddenly appears, the system must allocate S processors for it. Jobs can be created at any level of the hierarchy. The strategy used is for each manager to keep track of approximately how many workers below it are available (possibly several levels below it). If it thinks that a sufficient number are available, it reserves some number R of them, where R 2 S, because the estimate of available workers may not be exact and some machines may be down. If the manager receiving the request thinks that it has too few processors avail- able, it passes the request upward in the tree to its boss. If the boss cannot handle it either, the request continues propagating upward until it reaches a level that has enough available workers at its disposal. At that point, the manager splits the request into parts and parcels them out among the managers below it, which then do the same thing until the wave of scheduling requests hits bottom. At the bottom level, the processors are marked as "busy," and the actual number of processors allocated is re- ported back up the tree. To make this strategy work well, R must be large enough so that the probability is high that enough workers will be found to handle the whole job. Otherwise, the re- quest will have to move up one level in the tree and start all over, wasting considerable time and computing power. On the other hand, if R is too large, too many processors will be allocated, wasting computing capacity until word gets back to the top and they can be released. The whole situation is greatly complicated by the fact that requests for processors can be generated randomly anywhere in the system, so at any instant, multiple requests are likely to be in various stages of the allocation algorithm, potentially giving rise to out-of-date estimates of available workers, race conditions, deadlocks, and more. In Van, a mathematical analysis of the problem is given and various other aspects not de- scribed here are covered in detail.

3.4.4 Scheduling

The hierarchical model provides a general model for resource control but does not provide any specific guidance on how to do scheduling. If each process uses an entire processor (i.e., no multiprogramming), and each process is independent of all the others, any process can be assigned to any processor at random. However, if it is common that several processes are working together and must communicate frequently with each other, as in UNIX pipelines or in cascaded (nested) remote procedure calls, then it is desirable to make sure that the whole group runs at once. In this section we address that issue. Let us assume that each processor can handle up to N processes.

If there are plenty of machines and N is reasonably large, the problem is not finding a free machine (i.e., a free slot in some process table), but something more subtle. The basic difficulty can be illustrated by an example in which processes A and B run on one machine and processes C and D run on another. Each machine is time shared in, say, l00-millisecond time slices, with A and C running in the even slices, and B and D running in the odd ones, as shown in Figure 5a. Suppose that A sends many messages or makes many remote procedure calls to D. During time slice 0, A starts up and immediately calls D, which unfortunately is not running because it is now C's turn. After 100 milliseconds, process switching takes place, and D gets A's message, carries out the work, and quickly re- plies. Because B is now running, it will be another 100 milliseconds before A gets

the reply and can proceed. The net result is one message exchange every 200 milliseconds. What is needed is a way to ensure that processes that communicate frequently run simultaneously. Although it is difficult to determine dynamically the inter process communication patterns, in many cases a group of related processes will be started off together.

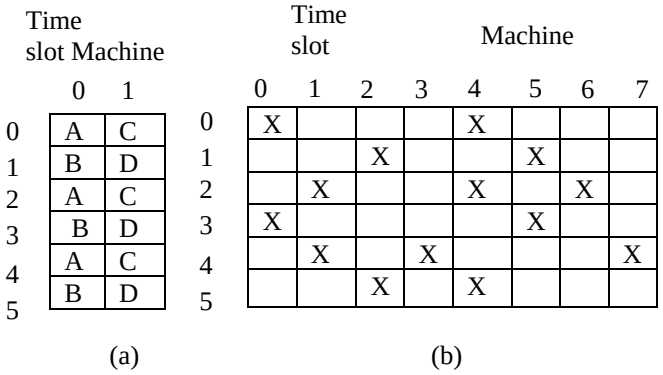

Figure 5. (a) Two jobs running out of phase with each other. (b) Scheduling matrix for eight machines, each with six time slots. The X's indicated allocated slots.

For example, it is usually a good bet that the filters in a UNIX pipeline will communicate with each other more than they will with other, previously started processes. Let us assume that processes are created in groups, and that intergroup communication is much more prevalent than intergroup communication. Let us further assume that a sufficiently large number of machines are available to handle the largest group, and that each machine is multiprogrammed with N process slots (N- way multiprogramming). Previous work has proposed several algorithms based on the concept of co- scheduling, which takes interprocess communication patterns into account while scheduling to ensure that all members of a group run at the same time. The first algorithm uses a conceptual matrix in which each column is the process table for one machine, as shown in Figure 5b. Thus, column 4 consists of all the processes that run on machine 4. Row 3 is the collection of all processes that are in slot 3 of some ma- chine, starting with the process in slot 3 of machine 0, then the process in slot 3 of machine 1, and so on. The gist of his idea is to have each processor use a round-robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, etc. A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized. By putting all the members of a process group in the same slot number, but on different machines, one has the advantage of N-fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication through- put. Thus in Figure 5b, four processes that must communicate should be put into slot 3, on machines 1, 2, 3, and 4 for optimum performance. This scheduling technique can be combined with the hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals. Ouster out also described several variations to this basic method to improve performance. One of these breaks the matrix into rows and concatenates the rows to form one long row. With k machines, any k consecutive slots belong to different machines. To allocate a new process group to slots, one lays a window k slots wide over the long row such that the leftmost slot is empty but the slot just outside the left edge of the window is full. If sufficient empty slots are present in the window, the processes are assigned to the empty slots; otherwise the window is slid to the right and the algorithm repeated. Scheduling is done by starting the window at the left edge and moving rightward by about one window's worth per time slice, taking care not to split groups over windows. Usterhout's paper discusses these and other methods in more detail and give some performance results.

3.4.5 Load Balancing

The goal of Usterhout's work is to place processes that work together on different processors, so that they can all run in parallel. Other researchers have tried to do precisely the opposite, namely, to find sub- sets of all the processes in the system that are working together, so that closely related groups of processes can be placed on the same machine to reduce inter process communication costs [30] [31] [32]. Yet other researchers have been concerned primarily with load balancing, to prevent a situation in which some processors are overloaded while others are empty [8] [38]. Of course, the goals of maximizing throughput, minimizing response time, and keeping the load uniform are to some extent in conflict, so many of the researchers try to evaluate different com- promises and trade-offs. Each of these different approaches to scheduling makes different assumptions about what is known and what is most important. The people trying to cluster processes to minimize communication costs, for example, assume that any process can run on any machine, that the computing needs of each process are known in advance, and that the interprocess communication traffic between each pair of processes is also known in advance. The people doing load balancing typically make the realistic assumption that nothing about the future behavior of a process is known.

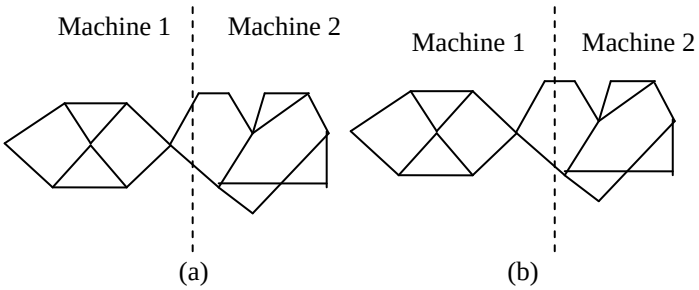

Figure 7. Two ways of statistically allocating processes (nodes in the graph) to machines. Arcs show which pairs of process communicate.

People making real systems, who care less about optimality than about devising algorithms that can actually be used, Let us now briefly look at each of these approaches.

Graph-Theoretic Models: If the system consists of a fixed number of processes, each with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair of processes, scheduling can be attacked as a graph-theoretic problem. The system can be represented as a graph, with each process a node and each pair of communicating processes connected by an arc labeled i^{th} the data rate between them. The problem of allocating all the processes to k processors then reduces to the problem of partitioning the graph into k disjoint sub graphs, such that each sub graph meets certain constraints (e.g., total CPU and memory requirements below some limit). Arcs that are entirely within one sub graph represent internal communication within a single processor (= fast), whereas arcs that cut across sub graph boundaries represent communication be- teen two processors (= slow). The idea is to find a partitioning of the graph that meets the constraints and minimizes the network traffic, or some variation of this idea. Figure 7a depicts a graph of interacting processors with one possible partitioning of the processes between two machines. Figure 7b shows a better partitioning, with less intermachine traffic, assuming that all the arcs are equally weighted. Many papers have been written on this subject [30] [31]. The results are somewhat academic, since in real systems virtually none of the as- assumptions (fixed number of processes with static requirements, known traffic matrix, error-free processors and communication) are ever met.

Heuristic Load Balancing: Here the idea is for each processor to estimate its own load continually, for processors to exchange load in- formation, and for process creation and migration to utilize this information. Various methods of load estimation are possible. One way is just to measure the number of runnable processes on each CPU periodically and take the average of the last n measurements as the load. Another way [20] is to estimate the residual running times of all the processes and define the load on a processor as the number of CPU seconds that all its processes will need to finish. The residual time can be estimated mostly simply by assuming it is equal to the CPU time already consumed. Bryant and Finkel also discuss other estimation techniques in which both the number of processes and length of remaining time are important. When round- robin scheduling is used, it is better to be competing against one process that needs 100 seconds than against 100 processes that each need 1 second. Once each processor has computed its load, a way is needed for each processor to find out how everyone else is doing. One way is for each processor to just broadcast its load periodically. After receiving a broadcast from a lightly loaded machine, a processor should shed some of its load by giving it to the lightly loaded processor. This algorithm has several problems. First, it requires a broadcast facility, which may not be available. Second, it consumes considerable bandwidth for all the "here is my load" messages. Third, there is a great danger that many processors will try to shed load to the same (previously) lightly loaded processor at once. A different strategy [8] is for each processor periodically to pick another processor (possibly a neighbor, possibly at random) and exchange load information with it. After the exchange, the more heavily loaded processor can send processes to the other one until they are equally loaded. In this model, if 100 processes are suddenly created in an otherwise empty system, after one ex- change we will have two machines with 50 processes and after two exchanges most probably four machines with 25 processes. Processes diffuse around the network like a cloud of gas. Actually migrating running processes is trivial in theory, but close to impossible in practice. The hard part is not moving the code, data, and registers, but moving the environment, such as the current position within all the open files, the current values of any running timers, pointers or file descriptors for communicating with tape drives or other I/O devices, etc. All of these problems relate to moving variables and data structures related to the process that are scattered about inside the operating system. What is feasible in practice is to use the load information to create new processes on lightly loaded machines, in- stead of trying to move running processes. If one has adopted the idea of creating new processes only on lightly loaded machines, another approach, called bidding, is possible [40]. When a process wants some work done, it broadcasts a re- quest for bids, telling what it needs (e.g., a 68000 CPU, 512K memory, floating point, and a tape drive). Other processors can then bid for the work, telling what their workload is, how much memory they have available, etc. The process making the request then chooses the most suitable machine and creates the process there. If multiple request-for-bid messages are outstanding at the same time, a processor accepting a bid may discover that the workload on the bidding machine is not what it expected because that processor has bid for and won other work in the meantime.

3.4.6 Distributed Deadlock Detection

Some theoretical work has been done in the area of detection of deadlocks in distributed systems. How applicable this work may be in practice remains to be seen. Two kinds of potential deadlocks are resource deadlocks and communication deadlocks. Re- source deadlocks are traditional deadlocks, in which all of some set of processes are blocked waiting for resources held by other blocked processes. For example, if A holds X and B holds Y, and A wants Y and B wants X, a deadlock will result. In principle, this problem is the same in centralized and distributed systems, but it is harder to detect in the latter because there are no centralized tables giving the status of all resources. The problem has mostly been studied in the context of database systems [39]. The other kind of deadlock that can occur in a distributed system is a communication deadlock. Suppose A is waiting for a message from B and B is waiting for C and C is waiting for A. Then we have a deadlock. [21] present an algorithm for detecting (but not preventing) communication deadlocks. Very crudely summarized, they assume that each process that is blocked waiting for a message knows which process or processes might send the message. When a process logically blocks, they assume that it does not really block but instead sends a query message to each of the processes that might send it a real (data) message. If one of these processes is blocked, it sends query messages to the processes it is waiting for. If certain messages eventually come back to the original process, it can conclude that a deadlock exists. In effect, the algorithm is looking for a knot in a directed graph.

4. DOS Survey

4.1 Comparison Criteria

The main goal of distributed file systems (DFS) or distributed operating systems (DOS) is to provide some level of transparency to the users of a computer network.

We have tried to develop a scheme -- referred to as a catalog of criteria -- that allows us to describe the systems in an implementation independent way. The main questions to be answered are: What kind of transparency levels are provided, how each kind of transparency achieved is what kind of communication strategy has been proposed and finally, does the distributed character of the system allow increased availability and reliability. The last question leads us to an analysis of replication schemes used and to an evaluation of proposed failure handling/recovery strategies.

4.1.1 Transparency Levels

We distinguish five levels of transparency. We speak of location transparency existing, when a process requesting a particular network resource does not necessarily know where the resource is located. Access transparency gives a user access to both local and remote located resources in the same way.

For reasons of availability, resources are sometimes replicated. If a user does not know whether a resource has been replicated or not, replication transparency exists.

The problem of synchronization is well-known. In a distributed environment this problem arises in an extended form. Encapsulating concurrency control inside a proposed system is what is meant by concurrency transparency. This includes schemes that provide system-wide consistency as well as weaker schemes in which user interaction can be necessary to recreate consistency. The distinction between transaction strategies and pure semaphore-based techniques is introduced in a special evaluation.

The last level of transparency is failure transparency. Network link failures or node crashes are always present within a network of independent nodes. Systems that provide stable storage, failure recovery and some state information are said to be failure transparent.

4.1.2 Heterogeneity

This survey furthermore gives information on the heterogeneity of the systems, i. e., assumptions made about the hardware and which operating system is used and whether that O.S. is a modified or look-alike version of another O.S.

We describe the underlying network. Is it a LAN, if so, what kind of LAN, or have gateways [7] been developed to integrate the systems into a WAN world.

4.1.3 Changes Made

There are two main forms of implementing distribution. First, a new layer can be inserted on top of an existing operating system that handles requests and provides remote access as well as some of the transparency levels.

Second, the distributed system can be implemented as a new kernel that runs on every node. This differentiation is a first hint of how portable or compatible a system is [28]. Some systems do not distribute all kernel facilities to all nodes. Dedicated servers can be introduced (strict client/server model). Some systems distribute a small kernel to all nodes and the rest of the utilities to special nodes (non-strict client/server model). Another group of systems are the so called integrated systems. In an integrated system each node can be a client, a server or both. This survey tries to describe these differences.

4.1.4 Communication Protocols

Message passing is the main form of communication (excepting multiprocessor systems which can use shared memory). We show which kind of protocols are used and describe specialized protocols if implemented.

4.1.5 Connection and RPC Facility

The kind of connection established by the (peer) communication partners is another important criteria. We distinguish between point-to-point connections (virtual circuits), datagram-style connections, and connections based on pipes or streams. If a remote procedure call (RPC) facility is provided we add this information as well.

4.1.6 Semantics

The users of a distributed system are interested in the way their services are provided and what their semantics are. We distinguish may-be (which means that the system guarantees nothing), at-least-once semantics (retrying to fulfill a service until acknowledged, sometimes done twice or more frequently), at-most-once semantics (mostly achieved by duplicate detection) and exactly-once semantics. The last kind is achieved by making a service an atomic issue (so called all-or-nothing principle).

4.1.7 Naming Strategy

We describe the naming philosophy and distinguish between object-oriented and traditional hierarchical naming conventions. Our overview includes the proposed name space itself as well as the mechanisms used to provide a system-spanning name space (e. g. mounting facilities or super root-approaches).

4.1.8 Security Issue

Security plays an important role within distributed systems, since the administration could possibly be decentralized and participating hosts cannot necessarily be trusted. Intruders may find it easy to penetrate a distributed environment. Therefore, sophisticated algorithms for encryption and authentication are necessary. We add four entries concerning this issue. First, encryption is used if no plain text will be exchanged over the communication media. Second, some systems make use of special hardware components to achieve security during the message transfer. Third, capabilities are provided that enable particular users access to resources in a secure and predefined way. Finally, we introduce the entry mutual authentication. This feature is provided if a sort of hand-shake mechanism is implemented that allows bilateral recognition of trustworthiness.

4.1.9 Failure handling

Failure handling/recovery is a very critical issue. Since some systems are designed to perform well in an academic environment and some systems are made highly reliable for

commercial use, trade-off decisions must be taken into account. We add the following four entries to our catalog of criteria. Does the system provide recovery after a client or a server crash, does it support orphan detection and deletion, and is there non-volatile memory called stable storage

4.1.10 Availability

Distributed systems can be made highly available by replicating resources or services among the nodes of the network. Thus, individual indispositions of nodes can be masked. (Nested) transactions are well-suited in a computer network. Our overview covers this feature. First of all, we look at the concurrency control scheme; i. e. availability is introduced through the following mechanisms: synchronization scheme, (nested) transaction facility, and replication.

4.1.11 Process Migration

Our final point of interest is process migration. Some object-oriented systems provide mobile objects; some traditional process-based systems support migration of processes. Sometimes, these approaches come along with load-balancing schemes to increase the system's performance. We include this issue in our survey.

Table 1: Table of comparison – Types of System & Transparency of Different Types of DOS's.

Name	Type of System		Transparency				
	DOS	DFS	location	access	replication	concurrency	failure
Accent	*		*	*			*
Alpine		*		*			
Amoeba	*		*	*		*	*
Andrew		*	*	*	*	*	
Argus	*		*	*			*
Athena		+	+	+			
BirliX	*		*	*	*	*	*
Cambridge DS	*		*	*			*
Cedar		*	*	*	*	*	*
Charlotte	*		*	*			
Chorus	*		*	*			
Clouds	*		*	*		*	*
Cosmos	*		*	*	+	+	+
Cronus	*		*	*	+		
DACNOS	+		*	*		*	
DEMOS/MP	*		*	*			*
DOMAIN	*		*	*			
DUNIX	*		*	*			
Eden	*		*	*	*		*
EFS		*	*	*			
Emerald	+		*	*			
GAFFES		*	*	*	*	*	
Grapevine			*	*	*		
Guide	*		*	*		*	
Gutenberg	*		*	*	*	*	*
HARKYS		*	*	*			
HCS	+		*	*			
Helix		*	*	*			*
IBIS		*	*	*	*	*	
JASMIN	*		*				
LOCUS	*		*	*	*	*	*
Mach	*		*	*			*
Medusa	*		*	+			
Meglos	*		+	+			
MOS	*		*	*			
NCA/NCS	+		*	*	*	*	*
Newcastle		*	*	*			
NEXUS		*	*	*			*
NFS		*	*	*			
Profemo	*		*	*		*	
PULSE	*		*	*	*		
QuickSilver	*		*	*			*
RFS		*	*	*			
Saguaro	*		*	*	+		
S-/F-UNIX		*		*			
SMB		+		*			
SOS	*		*	*			*
Sprite	*		*	*	*	*	
SWALLOW		*	*	*	*	*	*
V	*		*	*			
VAXcluster		*		*		*	*
XDFS		*	*	*		*	*

4.2 Table of Comparison

The table of comparison is given to summarize and compare the systems discussed. It should be viewed carefully, since in certain ways any categorized comparison can be misleading. However, this way an easily legible overview may be obtained. The table provides quick access to a large amount of highly condensed information. The entries are organized according to the criteria used to describe the systems. Sometimes, a similar issue or a comparable feature for an entry has been implemented. We mark this with a special symbol (+). Here Table 1 describes the types of system and transparency issues like replication, access, withstanding failures, etc. In the comparison, Cedar, Gutenberg, NCA/ NCS, Swallow performs well among all other DOS's [2] [13] [14] [16] [25] [27]. The comparison is given as follows

Table 2. Table of comparison – Hardware Requirements

Name	Heterogeneity	
	OS	CPUs
Accent		PERQ
Alpine	(Cedar)	Dorado
Amoeba	UNIX 4.2 BSD	68000s, PDP11, VAXes, IBM-PC, NS32016
Andrew	UNIX 4.2 BSD	SUN 2/3, MVAXes, IBM RT PC
Argus	Ultrix 1.2	MVAXes
Athena	UNIX	multi vendor HW
BirliX	UNIX 4.3 BSD	
Cambridge DS	TRIPOS	LSI-4, 68000, Z-80, PDP-11/45
Cedar		Alto, Dorado WS
Charlotte	UNIX	VAX-11/750
Chorus	UNIX V 3.2	SM 90, IBM AT PC, 68000s, 80386
Clouds	UNIX	VAX-11/750, SUN 3/60
Cosmos	UNIX	
Cronus	UNIX V7, 4.2 BSD, VMS	68000s, VAXes, SUNs, BBN C70
DACNOS	VM/CMS, PC DOS, VMS	VAXes, IBM PC, IBM/370
DEMOS/MP		Z8000
DOMAIN	UNIX III, 4.2 BSD	Apollo
DUNIX	UNIX 4.1 BSD	VAXes
Eden	UNIX 4.2 BSD	SUN, VAXes
EFS	MASSCOMP RT UNIX	MASSCOMP MP
Emerald	Ultrix	MVAX II
GAFFES	all kind	all kind
Grapevine	multiple OS	multi vendor HW
Guide	UNIX V	Bull SPS 7/9, SUN 3
Gutenberg	UNIX	
HARKYS	multiple UNIX systems	multi vendor HW
HCS	multiple os	multi vendor HW
Helix	XMS	68010-based
IBIS	UNIX 4.2 BSD	VAX-11/780
JASMIN	UNIX V	
LOCUS	UNIX 4.2 BSD, V.2	DEC VAX/750, PDP-11/45, IBM PC
Mach	UNIX 4.3 BSD	all VAXes, SUN 3, IBM PC, NS32016
Medusa	StarOS	Cm*
Meglos	UNIX	68000s, VAXes, PM-68k
MOS	UNIX V7	PDP-11, PCS/CADMUS 9000
NCA/NCS	DOMAIN/IX, UNIX 4.x BSD, VAX/VMS	SUNs, IBM PCs, VAXes
Newcastle	UNIX V7, III, BSD, Guts	PDP-11
NEXUS	UNIX 4.2 BSD	SUN
NFS	UNIX 4.2 BSD, V.2, VMS, SUN OS, Ultrix, MS/DOS	multi vendor HW
Profemo	UNIX 4.2 BSD	DEC VAXes
PULSE		LSI-11/23
QuickSilver		IBM RT-PC
RFS	UNIX V.3	multi vendor HW running UNIX V.3
Saguaro		SUN
S-/F-UNIX	UNIX V	DEC PDP-11
SMB	UNIX V, UNIX 4.x BSD, VMS;DOS 3.x	multi vendor PC
SOS		
Sprite	UNIX 4.x BSD	SUN 2/3
SWALLOW		
V	UNIX 4.x BSD	SUN 2/3, VAXstationII
VAXcluster	VAX/VMS	VAX 7xx-11s, VAX/8600
XDFS		Alto

Table 2 describes the hardware requirements of various DOS and supporting version types of OS they are using. Here most of the Dos are using UNIX as their supporting OS.

Table 3 describes the changes made the different types of protocols used for the communication. The communication part includes standard, specialized protocols, shared memory and RPC based protocols. And also it compares the connection types such as VC, datagram, Pipes/Streams of the different types of DOS. In the below comparison, Cronus, Mach, performs well again all the Dos in the case of new kernel [24] [26], shared memory etc.

Table 3. Table of comparison –Kernel, Communication and Connection

Name	Changes made		Communication				Connection		
	new kernel	new layer	standard protocols	specialized protocols	shared memory	RPC-based	VC	datagram	pipes/ streams
Accent	*			*				*	
Alpine	*					*			
Amoeba	*			Amoeba		*			
Andrew	*		UDP/IP			*		*	
Argus		*		ACP/IP				*	
Athena		+	TCP/IP				*		
BirliX	*				*	*		*	
Cambridge DS	*			*		*		*	*
Cedar		*	FTP				*		
Charlotte	*		*				*	*	
Chorus	*		*			*		*	
Clouds	*				*	*			
Cosmos	*								
Cronus	*		TCP, UDP	VLN		*	*	*	
DACNOS		*	OSI			*		*	
DEMOS/MP	*			*		*		*	
DOMAIN	*			*	*				
DUNIX	*		TCP/IP	*				*	
Eden		*				*			
EFS	*			RDP		*			
Emerald		*							
GAFFES		+	*			*	*		
Grapevine			*					*	*
Guide		*			*				
Gutenberg		*				*			
HARKYS		*	*			*			
HCS		*	*			*			
HARKYS		*	*			*			
HCS		*	*			*			
Helix	*		OSI				*		
IBIS		*	TCP/IP			*	*		
JASMIN	*			Paths					
LOCUS	*						*		
Mach	*			*	*	*		*	
Medusa	*				+		*		*
Meglos	*			*			*		
MOS	*		UDP/IP			*		*	
NCA/NCS		*	UDP/IP			*		*	
Newcastle		*				*			
NEXUS		*	*			*		*	
NFS	*		UDP/IP			*		*	
Profemo		*	UDP/IP		*	*			*
PULSE	*					*	*		
QuickSilver	*			*		*		*	
RFS	*		OSI TLI						*
Saguaro									*
S-/F-UNIX	*			KP			*		
SMB	*			SMB			*		
SOS	*			*		*		*	
Sprite	*			*		*			
SWALLOW		*		SMP				*	
V	*			VMTP		*		*	
VAXcluster	*			MSCP		*	*	*	
XDFS		*		Pup				*	

Table 4 describes the issues like semantics, naming and security. In the below comparison Amoeba, GAFFES, and Alphine performs well especially in the object oriented and Encryption related things.

Table 4. Table of comparison – Semantics, Naming and Security

Name	Semantics			Naming		Security			
	may be	at most once	exactly once	object-oriented	hier-archical	en-cryption	special HW	capa-bilities	mutual auth.
Accent		*		*				*	
Alpine		*			*	*		*	*
Amoeba		*		*		*	*	*	
Andrew	*				*				*
Argus			*	*					
Athena					*				
BirliX	*			*	*			*	
Cambridge DS	*	*		*				*	
Cedar		*			*				
Charlotte	*				*			*	
Chorus				*					
Clouds			*	*					
Cosmos		*		*				+	
Cronus	*			*				+	
DACNOS		*		*					*
DEMOS/MP	*		*		*			+	
DOMAIN				*	*				
DUNIX	*				*				
Eden		*		*				*	
EFS	*				*				
Emerald	*			*					
GAFFES			*		*	*	*	*	*
Grapevine					*				
Guide		*		*					
Gutenberg			*	*				*	
HARKYS	*				*				
HCS									
Helix				*		*		*	
IBIS	*				*				*
JASMIN	*				*			*	
LOCUS		*	*		*				
Mach		*		*				*	
Medusa				*					
Meglos					*				
MOS	*				*				
NCA/NCS		*		*					
Newcastle	*	*			*				
NEXUS	*		*	*	*				
NFS	*				*	*			
Profemo				*				*	
PULSE				*					
QuickSilver		*	*		*				
RFS	*				*				
Saguaro					*				
S-/F-UNIX	*				*				
SMB	*				*				
SOS	*	*		*				*	
Sprite		*			*				
SWALLOW		*		*				*	
V		*			*				
VAXcluster	*				*				
XDFS		*			*				

Name	Availability				Failures				
	synchro-nization	TA	nested TA	repli-cation	recovery client crash	recovery server crash	stable storage	orphan detection	Process Migration
Accent					+	+			*
Alpine	+	*			+	+			
Amoeba	*					*	*	*	
Andrew	*			*					
Argus	*	*	*	*	*	*	*	*	
Athena									
BirliX	*			*	*	*		*	*
Cambridge DS		+				*		+	
Cedar	*	*		*	*	*			
Charlotte						*			*
Chorus									*
Clouds	*	*	*		*	*	*		
Cosmos	*	*		*					
Cronus				*					*
DACNOS	*								
DEMOS/MP					*	*			*
DOMAIN									
DUNIX									
Eden		*		*		*			*
EFS		*			*	*			
Emerald	+								*
GAFFES	*	*	*	*					
Grapevine				*					
Guide	+	*	*						
Gutenberg	*	*		*			*		
HARKYS									
HCS									
Helix		*	*		*			+	
IBIS	*			*	*	*			
JASMIN									
LOCUS	*	*	*	*	*	*			*
Mach					+	+			*
Medusa	*								*
Meglos	+								
MOS				*					*
NCA/NCS	+			*					
Newcastle									
NEXUS		*	*						
NFS					*	*			
Profemo	*	*	*						
PULSE	*			*					
QuickSilver		*				*	*		
RFS	*				*				
Saguaro				*					
S-/F-UNIX									
SMB									
SOS					*	*			
Sprite	*			*					*
SWALLOW	*	*		*	*	*	*		
V									*
VAXcluster	*				*	*			
XDFS	*	*			*	*	*		

Table 5 describes about the comparison of availabity issues dealing with synchronization, Replication and the issues regarding failuires such as, recovery client crash, recovery server crash, stabe storage, orphan detection etc. In the below comparison, Amoeba, Argus, Cedar, Locus, Swallow, XDFS performs well among all the tghings specially in the issues like replication, recovery server crash, process migration.

Table 5. Table of comparison – Availability, Failures

5. Summary

Distributed operating systems are still in an early phase of development, with many unanswered questions and relatively little agreement among workers in the field about how things should be done. Many experimental systems use the client-server model with some form of remote procedure call as communication base, but there are also systems built on the connection model. Relatively little has been done on distributed naming, protection, and resource management, other than building straight-forward name servers and process servers. Fault tolerance is an up-and-coming area, with work progressing in redundancy techniques and atomic actions. Finally, a considerable amount of work has gone into the construction of file servers, print servers, and

various other servers, but here too there is much work to be done. The only conclusion that we draw is that distributed operating systems will be an interesting and fruitful area of research for a number of years to come.

References

[1] Adams, C. J., Adams, G. C., Waters, A. G., Leslie, I., and Kirk, P., "Protocol Architecture of the Universe Project," In Proceedings of the 6th International Conference on Computer Communication (London, Sept. 7-10). International Conference for Computer Communication, pp. 379- 383, 2001.

[2] Almes, G. T., Black, A. P., Lazowska, E. D., and Ni! Ie, J. D., "The Eden System: A Technical Review," IEEE Trans. Softw. Eng. Se-11 (Jan.). 43-59, 2006.

[3] Anderson, T., and Lee, P. A. Fault Tolerance, "Principles And Practice," Prentice-Hall International, London, 2000.

[4] Avizienis, A., and Chen, L., "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," In Proceedings of the International Computer Software and Applications Conference. IEEE, New York, pp. 149-155, 2008.

[5] Avizienis, A., and Kelly, J., "Fault Tolerance by Design Diversity," Computer 17 (Aug.), 66-80, 1984.

[6] Bal, H. E., Van Renesse, R., and Tanenbaum, A. S., "A Distributed, Parallel, Fault Tolerant Computing System," Rep. 1%106, Dept. of Mathematics and Computer Science. Vriie Univ., The Netherlands, Oct. 1999.

[7] Ball, J. E., Feldman, J., Low, R., Rashid, R., and Rovner, P., Rig, "Rochester's Intelligent Gateway: System Overview," IEEE Trans. Softw. Eng. Se-Z (Dec.), 321-329.

[8] Barak, A., and Shiloh, A. A., "Distributed Load-Balancing Policy for a Multicomputer," Softw. Pratt. Expert. 1.5 (Sept.), 901-913, 1985.

[9] Birman, K. P., and Rowe, L. A., "A Local Network Based on the Unix Operating System," IEEE Trans. Softw. Eng. Se-8 (Mar.), 137-146, 1982.

[10] Birrell, A. D.," Secure Communication Using Remote Procedure Calls," ACM Trans. Compute. Syst. 3, 1 (Feb.), 1-14, 1985.

[11] Birrell, A. D., and Needham, R. M., "A Universal File Server," IEEE Trans. Softw. Eng. Se-6, (Sept.), 450-453, 1980.

[12] Birrell, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls," ACM Trans. Compute. Syst. 2, 1 (Feb.), 39-59, 1984.

[13] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M., "Grapevine: An Exercise in Distributed Computing," Commun. Acm 25, 4 (Apr.), 260-274, 1982.

[14]Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M., "Experience with Grape-Vine: The Growth of a Distributed System. ACM Trans. Compute. Syst. 2, 1 (Feb.), 3-23, 1984.

[15] Black, A. P. "An Asymmetric Stream Communications System," Oper. Syst. Rev. (ACM) 17, 5, 4-10, 1983.

[16]Black, A. P., "Supporting Distributed Applications: Experience with Eden," In Proceedings of the 10th Symposium on Operating Systems Principles (Orcas Island, Wash., Dec. L-4). ACM, New York, pp. 181-193, 1985.

[17]Borg, A., Baumbach, J., and Glazer, S., "A Message System Supporting Fault Tolerance," Oper.Syst. Rev. (ACM) 17, 5, 90-99, 1983.

[18] Brown, M. R., Kolling, K. N. and Tag. E. A., "The Alnine File System," ACM Trans. Com- Put. Syst. 3, 4 ~Nov.), 261-293, 1985.

[19] Brownbridge, D. R., Marshall, L. F., Andrandell, B., "The Newcastle Connection-or Unixes of the World Unite! Softw," Pratt. Expert. 12 (Dec.), 1147-1162, 1982.

[20]Bryant, R. M., and Finkel, R. A., "A Stable Distributed Scheduling Algorithm," In Proceedings of the 2nd International Conference on Distributed Computer Systems (Apr.). IEEE, New York, pp. 314-323, 1981.

[21]Chandy, K. M., Misra, J., and Haas, L. M., "Distributed Deadlock Detection," ACM Trans. Compute. Syst. 1, 2 (May), 145-156, 1983.

[22]Cheriton, D. R.M, "The Thoth System: Multi- Process Structuring and Portability," American Elsevier, New York, 1982.

[23]Cheriton, D. R., "An Experiment Using Registers for Fast Message-Based Inter process Communication," Oper. Syst. Rev. 18 (Oct.), 12-20, 1984a.

[24]Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," IEEE Softw. 1 (Apr.), 19-42, 1984b.

[25]Cheriton, D. R., and Mann, T. P., "Uniform Access to Distributed Name Interpretation in the V System," In Proceedings of The 4th International ConferenceoOn Distributed Computing Systems. IEEE, New York, pp. 290-297, 1984.

[26] Cheriton, D. R., and Zwaenepoel, W., "The Distributed V Kernel and its Performance or Disk- Less Workstations," In Proceedings of the 9th Sym- Podium on Operating System Principles. ACM, New York, pp. 128-140, 1983.

[27] Cheriton, D. R., and Zwaenepoel, W., "One-to-Many Interprocess Communication in the V-System. In Szgcomm," '84 Tutorials and Svmposkm on Communications Architectures and Protocols (Montreal, Quebec, June 6-8). ACM, New York, 1984.

[28] Cheriton, D. R., Malcolm, M. A., Melen, L. S., and Sager, G. R. Thoth, "A Portable Real- Time Operating System," Commun. ACM 22, 2 (Feb.), 105-115, 1979.

[29]Chesson, G., "The Network Unix System," In Proceedings of The 5th Symposium on Operating Systems Principles," (Austin, Tex., Nov. 19-21). ACM, New York, pp. 60-66, 1975.

[30] Chow, T. C. K., and Abraham, J. A., "Load Balancing in Distributed Systems," IEEE Trans. Softw. Eng. Se-8 (July), 401-412, 1982.

[31] Chow, Y. C., and Kohler, W. H., "Models for Dynamic Load Balancing in Heterogeneous Multiplex Processor Systems," IEEE Trans. Compute. C-28 (May), 354-361, 1979.

[32] Chu, W. W., Holloway, L. J., Min-Tsung, L., and Efe, K., "Task Allocation in Distributed Data Processing," Computer 23 (Nov.), 57-69, 1980.

[33]Curtis, R. S., and Wi~Ie, L. D., Global Naming In Distributed Systems. IEEE Softw. 1, 76-80, 1984.

[34] Dalal, Y. K., "Broadcast Protocols in Packet Switched Computer Networks," Ph.D. Dissertation, Computer Science Dept., Stanford Univ., Stan- Ford, Calif.

[35] Dellar, C., " A File Server for a Network of Low-Cost Personal Microcomputers," Softw. Pratt. Erper. 22 (Nov.), 1051-1068, 2009.

[36]Dennis, J. B., and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations. Commun," ACM 9, 3 (Mar.), 143- 154, 2009.

[37] Dion, J, "The Cambridge File Server," Oper. Syst. Reu. (ACM) 14 (Oct.), 41-49.

[38] Efe, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems," Computer 15 (June), 50-56, 1992.

[39]Eswaran, K. P., Gray, J. N., Lorie, J. N., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System," Com- Mun. ACM 19, 11 (Nov.), 624-633, 1986.

[40]Farber, D. J., and Larson, K. C., "The System Architecture of the Distributed Computer System-The Communications System," In Proceedings of the Symposium on Computer Networks (Brooklyn, Apr.). Polytechnic Inst. of Brooklyn, Brooklyn, N.Y, 1972.