RESEARCH ISSUES IN DISTRIBUTED OPERATING SYSTEMS

Andrew S. Tanenbaum
Robbert van Renesse

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

As distributed computing becomes more widespread, both in high-energy physics and in other applications, centralized operating systems will gradually give way to distributed ones. In this paper we discuss some current research on five issues that are central to the design of distributed operating systems: communications primitives, naming and protection, resource management, fault tolerance, and system services. For each of these issues, some principles, examples, and other considerations will be given.

1. INTRODUCTION

As distributed computing becomes more widespread, both in high-energy physics and in other applications, centralized operating systems will gradually give way to distributed ones. Distributed operating systems have many points in common with centralized ones, but they also have many distinctive features of their own. They often differ in the areas of how processes communicate with each other, how resources are named (and protected), how they are managed, how robustness (fault tolerance) is achieved, and how system services are provided to user processes. In the following sections we will survey current research ideas in each of these areas.

2. COMMUNICATION PRIMITIVES

The computers forming a distributed system normally do not share primary memory, so communication via shared memory techniques such as semaphores are generally not applicable. Instead, message passing in one form or another is used. One widely discussed framework for message-passing systems is the ISO OSI reference model, which has seven layers, each performing a well-defined function [Zimmerman 1980]. The seven layers are: physical layer, data-link layer, network layer, transport layer, session layer, presentation layer, and application layer. 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 very substantial. Nearly all the experimental distributed systems discussed in the literature so far have opted for a radically different, and much simpler model, so we will not mention the ISO model further in this paper.

2.1. 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 file) sends a message to the server and then waits for a reply message. In the most naked form, the system just provides two primitives: SEND and RECEIVE. The SEND primitive specifies the destination and provides a buffer; 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 teardown is required.

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 vs. reliable and nonblocking vs. blocking primitives. At one extreme, SEND can put a message out onto the network and wish it good luck. No guarantee of delivery is provided, and no retransmissions are attempted by the system. At the other extreme, the SEND can handle lost messages, retransmissions, and acknowledgements internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

The other choice is between nonblocking 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 it is 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. The advantage of these nonblocking primitives is that they provide the maximum flexibility: programs can compute and perform message I/O in parallel any way they want to.

Nonblocking 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 acknowledgement received (reliable blocking primitive). Either way, the program may immediately modify the buffer without danger.

Another design decision that is closely related to the ones above is whether or not to buffer messages. The simplest strategy is not to buffer. When a sender has a message for a receiver that has not (yet) executed a RECEIVE primitive, the sender is blocked until a RECEIVE has been done, at which time the message is copied from sender to receiver. This strategy is sometimes referred to as a **rendezvous**, and it provides for simple flow control.

A more structured form of communication is achieved by distinguishing requests from replies. With this approach, one typically has three primitives: REQREP, GETREQUEST, and SENDREPLY. REQREP is used by clients to send requests and get replies. It combines a SEND to a server combined with a RECEIVE to get the server's reply. GETREQUEST 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 SENDREPLY. By thus restricting the message traffic, and by using reliable, blocking primitives, one can create some order in the chaos.

2.2. Remote Procedure Call

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 [Birrell and Nelson 1984]. The idea is to make the semantics of intermachine communication as similar as possible to normal procedure calls because the latter is familiar, 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 REQREP, GETREQUEST, SENDREP primitives, with a more user-friendly syntax.

A nice way of organizing remote procedure call goes something like this. 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 well 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 REQREP, and blocks waiting for an answer.

At the remote machine, another stub procedure should be waiting for a message using GETREQUEST. 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 in fact called locally, 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 lurk 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 it goes. 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, one can envision some kind of capability mechanism [Dennis and Van Horn 1966] being set up, with capabilities being used as pointers. For small objects, such as integers and booleans, the amount of overhead and mechanism needed to create a capability and send it in a protected way is so large that it is highly undesirable.

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 give problems between 1's and 2's complement machines.

3. NAMING AND PROTECTION

All operating systems support objects such as files, directories, segments, mailboxes, processes, services, servers, nodes, and I/O devices that need to be named in order to use them. When a process wants to access some object, it must present some kind of name to the operating system to describe 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.

3.1. Naming as Mapping

Naming 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.

Another example of naming is the mapping of virtual addresses onto physical addresses in a virtual memory system. The paging hardware takes a virtual address as input, and yields a physical address as output 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 it that process is running. The mapping need not always be unique, for example, if there are multiple processes prepared to offer the same service.

3.2. Name Servers

In centralized systems, the problem of naming can be effectively handled in a straightforward way. The system maintains a table or data base providing the necessary name-to-object mappings. The most straightforward generalization of this approach to distributed systems is the single name server model. This model entails having a server that 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 data base is built up by having services, processes, etc. that want to be publicly known registering with the name server. 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 geographically distributed system, perhaps with nodes in different cities or even countries, having a single name server will be inefficient due 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 gateways 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 local network where the object is located so the mapping can be done. Telephone numbers use such a hierarchy, composed of country code, area code, exchange code (first 2 or3 digits of telephone number) and subscriber line number.

A more extreme way of distributing the name server is to have each machine manage its own names. To look up a name, one broadcasts it on the network. At each machine 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 [Dalal 1977].

Although the normal use of a name server is to map an ASCII string onto a binary number used internal 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 (hardwired) 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 it is supposed to be offering so it can figure out what program to reboot.

4. RESOURCE MANAGEMENT

Resource management in a distributed system differs from that in a centralized system in a fundamental way. Centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed systems do not. The problem of managing resources without having accurate global state information is very difficult. Relatively little work has been done in this area. In the following sections we will look at some work that has been done on distributed process management and scheduling.

4.1. Process Management

One of the key resources to be managed in a distributed system is the set of available processors. If these machines are multiprogrammed (time-shared), with N process table slots per machine, each physical processor can be regarded as N virtual processors to be managed. 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 [Wittie and van Tilborg 1980]. 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.

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, or in a more autocratic system, by the sick manager’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 gives up the ghost, the remaining members promote someone one level down as replacement.

While this scheme is not completely distributed, it is feasible, and in practice works well. In particular, the system is self-repairing, and can survive occasional crashes of both workers and managers without any long-term effects.

4.2. 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 the whole group runs at once. We will first look at one scheduling strategy proposed for monoprogrammed systems, and then at one for multiprogrammed systems.

In MICROS the processors are not multiprogrammed, 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 described earlier. 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 >= 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 available, it passes the request upwards 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 reported back up the tree.

To make this strategy work well, R must be chosen large enough that the probability is high that enough workers will be found to handle the whole job. Otherwise the request 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 chosen too large, too many processors will be allocated, wasting computing capacity until word gets back to the top and they can be released. In [Van Tilborg and Wittie 1981] a mathematical analysis of the problem is given and various other aspects not described here are covered in detail.

Ousterhout [1982] 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. 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 machine, 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 throughput. 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.

Ousterhout 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. Ousterhout's paper discusses these and other methods in more detail and gives some performance results.

A completely different approach to scheduling that does not require any tables at all is **bidding** [Farber and Larson 1972]. When a process (e.g., a command interpreter) has a need to create a new process in order to run some program, it broadcasts a requests for bids on the network, describing what kind of processor it needs or what program it wants run. All available processors are normally enabled to listen for such requests. When an available processor sees a request-for-bid message, it checks to see if it has the appropriate configuration (e.g., floating point hardware, enough memory), and if so, makes a bid. The requesting process evaluates the bids and then selects one or more bidders to run the new process(es).

This strategy can be generalized to multiprogrammed processors by having the bids tell how much CPU capacity the bidder is willing to divert to the new process if it wins the bid. A heavily loaded processor will obviously be able to devote less capacity to a new process than a lightly loaded one. Therefore lightly loaded processors will usually win until they get too much work, at which time they will begin losing. This scheme thus provides for an automatic, and fully dynamic, form of load balancing.

5. FAULT TOLERANCE

In the past few years, two approaches to making distributed systems fault tolerant have emerged. They differ radically in orientation, goals, and attitude toward the theologically sensitive issue of the perfectability of mankind (programmers in particular). One approach is based on redundancy and one is based on the notion of an atomic transaction. Both are described briefly below.

5.1. Redundancy Techniques

All the redundancy techniques take advantage of the existence of multiple processors by duplicating critical processes on two or more machines. A particularly simple, but effective, technique is to provide every process with a backup process on a different processor. All processes communicate by message passing. Whenever anyone sends a message to a process, it also sends the same message to the backup process. The system insures that neither the primary nor the backup can continue running until it has been verified that both have correctly received the message.

The fault tolerance comes from the property that if one process crashes due to any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, thus making a new backup to maintain the fault tolerance in the future. Borg et al. [1983] have described a system using these principles.

One disadvantage of duplicating every process is the extra processors required, but another, more subtle problem, is that if processes exchange messages at a high rate, a considerable amount of CPU time may go into keeping the processes synchronized at each message exchange. Powell and Presotto [1983] have described a redundant system that puts almost no additional load on the processes being backed up. In their system, all messages sent on the network are recorded by a special "recorder" process. From time to time, each process checkpoints itself onto a remote disk.

If a process crashes, recovery is done by sending the most recent checkpoint to an idle processor and telling it to start running. The recorder process then spoon feeds it all the messages that the original process received between the checkpoint and the crash. Messages sent by the newly restarted process are discarded. Once the new process has worked its way up to the point of crash, it begins sending and receiving messages normally, without help from the recording process.

Both of the above techniques are only applicable to making the system tolerant of hardware errors. However, it is also possible to use the redundancy inherent in distributed systems to make systems tolerant of software errors as well. One approach is to structure each program as a collection of modules, each one with a well-defined function and a very precisely specified interface with all the others. Instead of writing a module only once, N programmers are asked to program it, yielding N functionally identical modules.

During execution, the program runs on N machines in parallel. After each module finishes, the machines compare their results and vote on the answer. If a majority of the machines say that the answer is X , then all of them use X as the answer, and all continue in parallel with the next module. In this manner, the effects of an occasional software bug can be voted down. If formal specifications for any of the modules are available, the answers can also be checked against the specifications to guard against the possibility of accepting an answer that is clearly wrong. Some work in this area is discussed in [Avizienis and Kelly 1984]

5.2. Atomic Transactions

When multiple users spread over several machines are concurrently updating a distributed data base, and one or more machines crash, the potential for chaos is truly impressive. In a certain sense, the current situation is a step backwards from the technology of the 1950s. The normal way of updating a data base then was to have one magnetic tape, called the "master file," and one or more tapes with updates (e.g., daily sales reports from all of a company's stores). The master tape and the updates were brought to the computer center, which then mounted the master tape and one update tape, and ran the update program to produce a new master tape. This new tape was then used as the "master" for use with the next update tape.

This scheme had the very real advantage that if the update program crashed, one could always fall back on the previous master tape and the update tapes. In other words, an update run could be viewed as either running correctly to completion (and producing a new master tape), or having no effect at all (crash part way through, new tape discarded). Furthermore, update jobs from different sources always ran in some (undefined) sequential order. It never happened that two concurrent users would read a field in a record, (e.g. 6), each add 1 to the value, and then each store a 7 into that field, instead of the first one storing a 7 and the second one storing an 8.

The property of run-to-completion or do-nothing is called an **atomic update**. The property of not interleaving two jobs is called **serializability**. The goal of the people working on the atomic transaction approach to fault tolerance has been to regain the advantages of the old tape system without giving up the convenience of data bases on disk that can be modified in place, and furthermore to be able to do everything in a distributed way.

Lampson [1981] has described a way of achieving atomic transactions by building up a hierarchy of abstractions. We will summarize his model below. Real disks can crash during READ and WRITE operations in unpredictable ways. Furthermore, even if a disk block is correctly written, there is a small, but nonzero probability of it subsequently being corrupted by newly developed bad spot on the disk surface. The model assumes that spontaneous block corruptions are sufficiently infrequent that the probability of two such events happening within some predetermined time, T , is negligible. To deal with real disks, the system software must be able to tell if a block is valid or not, for example, by using a checksum.

The first layer of abstraction on top of the real disk is the "careful disk," in which every

CAREFUL-WRITE is read back immediately to verify that it is correct. If the CAREFUL-WRITE persistently fails, the system marks the block as "bad" and then intentionally crashes. Since CAREFUL-WRITEs are verified, CAREFUL-READs will always be good, unless a block has gone bad after being written and verified.

The next layer of abstraction is **stable storage**. A stable storage block consists of an ordered pair of careful blocks, typically corresponding blocks on different drives to minimize the chance of both being damaged by a hardware failure. The stable storage algorithm guarantees that at least one of the blocks is always valid. The STABLE-WRITE primitive first does a CAREFUL-WRITE on one block of the pair, and then the other. If the first one fails, a crash is forced, as mentioned above, and the second one is left untouched.

After every crash, and at least once every time T , a special cleanup process is run to examine each stable block. If both blocks are "good" and identical, nothing has to be done. If one is "good" and one is "bad" (failure during a CAREFUL-WRITE), the "bad" one is replaced by the "good" one. If both are "good" but different (crash between two CAREFUL-WRITEs), the second one is replaced by a copy of the first one. This algorithm allows individual disk blocks to be updated atomically and survive infrequent crashes.

Stable storage can be used to create "stable processors" [Lampson 1981]. To make itself crashproof, a CPU must checkpoint itself on stable storage periodically. If it subsequently crashes, it can always restart itself from the last checkpoint. With stable storage and stable processors, processes can implement atomic transactions by first writing an intentions list to stable storage and then writing a commit flag to stable storage.

6. SERVICES

In a distributed system it is natural to take functions that have traditionally been provided by the operating system and have them be offered by user-level server processes. This approach leads to a smaller (hence more reliable) kernel and makes it easier to provide, modify and test new services. In the following sections we will look at some of these services.

6.1. File Service

Without much doubt, the most important service in any distributed system is the file service. Many file services and file servers have been designed and implemented, so a certain amount of experience is available [e.g., Birrell and Needham 1980; Fridrich and Older 1981; Sturgis et al. 1980; Swinehart et al. 1979]. One thing that is clear from this experience is that there are many different ways to make a file server.

Conceptually, one can isolate three components that a file system normally has:

- Disk service
- Flat file service
- Directory service

The disk service is concerned with reading and writing raw disk blocks, without regard to how they are organized. A typical command to the disk service is to allocate and write a disk block, and return a capability or address (suitably protected) for the block written so it can be read later.

The flat file service is concerned with providing its clients with an abstraction consisting of files, each of which is a linear sequence of records, possibly 1-byte records as in UNIX, or possibly client-defined records. The operations here are reading and writing records, starting at some particular place in the file. The client need not be concerned with how or where the data in the file are stored.

The directory service provides a mechanism for naming and protecting files, so they can be accessed conveniently and safely. The directory service typically provides objects called directories that map ASCII names onto the internal identification used by the file service.

One important issue in a distributed system is how closely these three elements are integrated. At one extreme, the system can have distinct disk, file and directory services that run on different machines and only interact via the official interprocess communication mechanism, such as remote procedure call. This approach is the most flexible, because anyone needing a different kind of file service (e.g., a B-tree file) can use the standard disk server. It is also potentially the least efficient, since it generates considerable inter-server traffic.

At the other extreme, there are systems in which all three functions are handled by a single program, typically running on a machine to which a disk is attached. With this model, any application that needs a slightly different file naming scheme is nevertheless forced to start all over making its own private disk server. The gain, however, is increased runtime efficiency, because the disk, file and directory services do not have to communicate over the network.

In addition to the question of how the file system is split up into pieces, there are a variety of other issues that the designers of a distributed file system must address. For example, will the file service be virtual-circuit oriented or connectionless. In the former approach, before reading a file, the client must do an OPEN on the file, at which time the file server fetches some information about the file (in UNIX terms, the i-node) into memory, and the client is given some kind of a connection identifier. This identifier is used in subsequent READs and WRITEs. In the connectionless approach, each READ request identifies the file and file position in full, so the server need not keep the i-node in memory (although most servers will maintain a cache for efficiency reasons). The virtual-circuit model fits in well with the ISO OSI 7 layer reference model view of the world, and the connectionless model fits in well with the remote procedure call model of the world.

6.2. Print Service

Nearly all distributed systems have some kind of print service, to which clients can send files or file names or capabilities for files with instructions to print them on one of the available printers, possibly with some text justification or other formatting beforehand. In some cases the whole file is sent to the print server in advance, thus requiring the server to buffer it. In other cases only the file name or capability is sent, with the print server reading the file block by block as needed. The latter strategy eliminates the need for buffering (read: a disk) on the server side, but can cause problems if the file is modified subsequent to the time the print command is given but prior to the actual printing. Users generally prefer "call by value" rather than "call by reference" semantics for printers.

6.3. Process Service

Every distributed operating system needs some mechanism for creating new processes. At the lowest level, deep inside the system kernel, there must be a way of creating a new process from scratch. One way is to have a FORK call, as UNIX does, but other approaches are also possible. For example, in Amoeba, it is possible to ask the kernel to allocate chunks of memory of given sizes. The caller can then read and write these chunks, loading them with the text, data, and stack segments for a new process. Finally, the caller can give the filled in segments back to the kernel and ask for a new process built up from these pieces.

At a higher level, it is frequently useful to have a process server that one can ask if there is a Pascal or troff or some other service somewhere in the system. If there is, the request is forwarded to the relevant server. If there is not, it is the job of the process server to build a process somewhere and give it the request. After, say, a VLSI design rule checking server has been created and has done its work, it may or may not be a good idea to keep it in the machine where it was created, depending on how much work (e.g., network traffic) is required to load it, and how often it is called. The process server could easily manage a server cache on a least recently used basis, so that servers for common applications are usually preloaded and ready to go. As special-purpose VLSI processors become available for compilers and other applications, the process server should be given the job of managing them in a way transparent to the system's users.

6.4. Terminal Service

How the terminals are tied to the system obviously depends on the system architecture to a large extent. If the system consists of a small number of minicomputers, each with a well-defined and stable user population, then each terminal can be hardwired to the computer its user normally logs on to. If however, the system consists entirely of a pool of processors that are dynamically allocated as needed, it is better to connect all the terminals to one or more terminal servers that serve as concentrators.

The terminal servers can also provide such features as local echoing, intraline editing, and window management, if desired. Furthermore, the terminal server can also hide the idiosyncracies of the various terminals in use by mapping them all onto a standard virtual terminal.

6.5. Mail Service

Electronic mail is a popular application of computers these days. Practically every university computer science department in the Western world is on at least one international network for sending and receiving electronic mail. When a site consists of only one computer, keeping track of the mail is easy. However, when a site has dozens of computers spread over multiple local networks, users often want to be able to read their mail on any machine they happen to be logged on to. This desire gives rise to the need for a machine-independent mail service, rather like a print service that can be accessed system wide.

6.6. Time Service

There are two ways one can organize the time service. In the simplest way, clients can just ask the service what time it is. In the other way, the time service can broadcast the correct time periodically, to keep all the clocks on the other machines in sync. The time server can be equipped with a radio receiver tuned to WWV or some other transmitter that provides the exact time down to the microsecond.

6.7. Boot Service

The boot service has two functions: bringing up the system from scratch when the power is turned on, and helping important services survive crashes of their servers. In both cases it is helpful if the boot server has a hardware mechanism for forcing a recalcitrant machine to jump to a program in the balky machine's own ROM, in order to reset it. The ROM program could simply sit in a loop waiting for a message from the boot service. The message would then be loaded into that machine's memory and executed as a program.

The second function alluded to above is the "immortality service." An important service could register with the boot service, which would then poll it periodically to see if it were still functioning. If not, the boot service could initiate measures to patch things up, for example, forcibly reboot it or allocate another processor to take over its work. To provide high reliability, the boot service should itself consist of multiple processors, each of which keeps checking that the other ones are still working properly.

7. SUMMARY

The majority of current research on distributed operating systems focuses on the remote procedure call as the basic communication primitive. Naming is usually done by specialized name servers, often hierarchically organized. Resource management in a distributed system is very much an open subject, with a few scheduling and allocation algorithms known, but much work remaining to be done. Fault tolerance can be approached either using redundancy techniques, or using atomic actions. Finally, many kinds of servers can be incorporated into a distributed system to move traditional operating system functions out of the kernel and into user space, including the file system, process management, and terminal management.

8. REFERENCES

Avizienis, A., and Kelly, J. "Fault Tolerance by Design Diversity," *Computer*, vol. 17, pp. 66-80, Aug. 1984.

Birrell, A.D., and Needham, R.M. "A Universal File Server," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 450-453, Sept. 1980.

Birrell, A.D. and Nelson, B.J. "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp. 39-59, Feb. 1984.

Dalal, Y.K. "Broadcast Protocols in Packet Switched Computer Networks," Ph. D. Thesis, Stanford Univ., 1977.

Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, pp. 143-154, March 1966.

Farber, D.J. and Larson, K.C. "The System Architecture of the Distributed Computer System-The Communications System," *Symp. on Computer Netw.*, Polytechnic Institute of Brooklyn, April 1972.

Fridrich, M. and Older, W. "The Felix File Server," *Proc. 8th Symp. on Operating Syst. Prin.*, ACM, pp. 37-44, 1981.

Lampson, B.W. "Atomic Transactions," in *Distributed Systems - Architecture and Implementation*, Berlin: Springer-Verlag, pp. 246-265, 1981

Ousterhout, J.K. "Scheduling Techniques for Concurrent Systems," *Proc. 3rd Int'l Conf. on Distributed Computing Syst*, IEEE, pp. 22-30, 1982.

Powell, M.L., and Presotto, D.L. "Publishing-A Reliable Broadcast Communication Mechanism," *Proc. 9th Symp. on Operating Syst. Prin.*, ACM, pp. 100-109, 1983.

Sturgis, H.E., Mitchell, J.G., and Israel, J. "Issues in the Design and Use of a Distributed File System," *Op. Syst. Rev.*, vol. 14, pp. 55-69, July 1980.

Swinehart, D., McDaniel, G., Boggs, D. "WFS: A Simple Shared File System for a Distributed Environment," *Proc. 7th Symp. on Operating Syst. Prin.*, ACM, pp. 9-17, 1979.

Van Tilborg, A.M. and Wittie, L.D. "Wave Scheduling: Distributed Allocation of Task Forces in Network Computers," *Proc. 2nd Int'l Conf. on Distributed Computing Syst.*, IEEE, pp. 337-347, 1981.

Wittie, L.D. and van Tilborg, A.M. "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Trans. Comp.*, pp. 1133-1144, Dec. 1980.

Zimmerman, H. "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Comm.*, vol. COM-28, pp. 425-432, April 1980.