RELIABILITY ISSUES IN DISTRIBUTED OPERATING SYSTEMS†

Andrew S. Tanenbaum
Robbert van Renesse

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

ABSTRACT

Distributed systems span a wide spectrum in the design space. In this paper we will look at the various kinds and discuss some of the reliability issues involved. In the first half of the paper we will concentrate on the causes of unreliability, illustrating these with some general solutions and examples. Among the issues treated are interprocess communication, machine crashes, server redundancy, and data integrity. In the second half of the paper, we will examine one distributed operating system, Amoeba, to see how reliability issues have been handled in at least one real system, and how the pieces fit together.

1. INTRODUCTION

It is difficult to get two computer scientists to agree on what a distributed system is. Rather than attempt to formulate a watertight definition, which is probably impossible anyway, we will divide these systems into three broad categories:

- Closely coupled systems
- Loosely coupled systems
- Barely coupled systems

The key issue that distinguishes these systems is the grain of computation, which can be roughly expressed as the computation time divided by the communication time. If this ratio is below 10, we have a closely coupled system. If it is between 10 and 100 we have a loosely coupled system. Above 100 the system is barely coupled.

In practice, the amount of time required for communication is determined by the communication hardware and the operating system. In a system consisting on a large number of CPU boards on a single backplane with shared memory, it may be possible for one processor to write a word in another processor's memory in microseconds. On the other hand, processors that communicate over a local area network by message passing typically require milliseconds to send a message and get a reply. Finally, when a wide-area network is being used, communication times of hundreds of milliseconds or more are normal.

These hardware parameters tend to give rise to three kinds of distributed systems, each with their own properties. These systems differ in terms of how the users view the system, how much autonomy the individual processors have, how problems are partitioned among the processors, how work migrates among the processors, how the load is balanced, how

† This work was supported in part by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.)

interprocess communication is done, whether the system is homogeneous or heterogeneous, and finally how reliable the total system is and what the failure modes are.

At one extreme we have closely-coupled multiprocessors with shared memory communicating over a backplane type bus with short bursts of computation interleaved with short bursts of communication. This is fine-grained parallelism.

Usually all the processors used in this kind of system are identical and fairly close together (same room). Frequently, all the processors are working together on a single problem. Although the system designers may try to make the presence of multiple processors transparent, with hundreds or thousands of CPUs it may be difficult to keep all the processors busy unless the parallelism is programmed explicitly.

The second kind of system is the loosely coupled system, typically consisting of a number of workstations or personal computers communicating over a local area network. In some systems a rack of processors is present, any or all of which can be dynamically allocated as the need arises. In some cases, the user perceives of the system as a collection of autonomous computers that share a common file server or printer. In other cases, the system looks like a virtual uniprocessor. In other words, to the user, the whole system looks like a traditional multiuser time sharing system, rather than a network of independent machines.

There are two general approaches that can be used in such systems. In the first one, all the machines run the same operating system. In the second one, different machines can run different native operating systems, with a layer of software on top to make them look (more) homogeneous. A general survey on distributed systems is given by Tanenbaum and van Renesse[1].

The third kind of system consists of (typically large) computers or local area networks connected by a low-bandwidth, wide-area network. These machines are barely connected in the sense that communication costs normally dominate the computation costs. Still, for some applications, such as doing joins in a database system, the amount of computing is so large that the system can be made to appear to the user as a single system, despite the low-bandwidth connection between the pieces.

A key point that is common to all these systems, however, is the question of whether the parallelism provided by the multiple processors is implicit or explicit. This point has important implications for reliability aspects of the system. If the system looks to the user like a virtual uniprocessor, there is relatively little that can be done about reliability at the user level. The reliability must be handled by the system. On the other hand, if users can explicitly control the parallelism, it is possible for them to use the redundancy to enhance the reliability.

A simple example may make this point clear. Some distributed file systems offer atomic transactions[2] as a primitive operation. The user can specify that a transaction be started, issue commands to read and write files, and then commit the transaction. The system then either runs the entire transaction to completion, or fails, leaving all the files in their original state. Such a file system may well use multiple processors and multiple disks internally, but there is nothing the users can do to influence the reliability behavior.

Now consider a different example, a system with a rack of processors that can be dynamically allocated to processes upon request. A process can request *n* processors, set all of them working on the same problem (possibly with different algorithms), and then accept the majority answer when all have reported back. In this system the parallelism is explicit, so the user can decide how much redundancy is required for the problem at hand. The conclusion is that systems with explicit parallelism tend to be more flexible, but require more work on the part of the user.

2. CAUSES OF UNRELIABILITY

Space limitations prevent us from examining the reliability aspects of all three kinds of systems in detail, so we will focus primarily on the middle category—loosely coupled systems. In particular, in this section we will look at some problems that cause systems to be unreliable and on some of the solutions that have been proposed for these problems. In the next section we will look at one distributed system, Amoeba, to see how these problems have been attacked and how the various components fit together to make a more reliable system.

2.1. Interprocess Communication

When the processors in a distributed system are connected by a thin wire local network, interprocess communication primitives that explicitly or implicitly require shared memory (such as semaphores), are not desirable.

Instead some form of message passing is needed. One widely discussed framework for message-passing in computer networks is the ISO-OSI model[3]. To make a long story short, the various protocols that go with this model are so complex and cumbersome, that their use in a high performance local-area network is unattractive at best.

The model favored by most researchers in this area is the client-server model, in which a client wanting some service (e.g., a block from a file) sends a message to the server, which then sends a reply. The basic primitives in the simplest form of client-server model are SEND and RECEIVE, each specifying an address (destination or source), and a buffer.

These primitives come in several varieties. First of all, there is the question of whether transmission is reliable or not. On some systems, SEND means put the message out onto the network and hope for the best. Processes needing better reliability than that must arrange for it themselves. Other systems use low-level protocols that do automatic timeout and retransmission. Here we see a clear tradeoff between performance and reliability.

A second question is blocking vs. nonblocking primitives. With a blocking SEND, the sender is suspended until the message has been transmitted (unreliable transmission) or transmitted and acknowledged (reliable transmission). With a nonblocking SEND, the sender continues immediately. If the sender modifies the buffer, these changes may or may not be transmitted, depending on whether transmission has taken place or not. Similarly, a blocking RECEIVE waits until a message arrives, but a nonblocking RECEIVE merely provides a buffer. When a message arrives, the receiver gets an interrupt. Nonblocking primitives are harder to use (hence less reliable) but provide more parallelism and higher performance.

Based on experience, many system designer have decided to favor reliability over performance, which has led to the **remote procedure call**[4-6]. In this scheme, the client makes what looks like a call to a procedure running on the server's machine, but it actually makes a call to a **stub procedure** running on its own machine, as shown in Fig. 1. The stub procedure packages all the parameters in a message, which it then reliably sends to a stub on the server's machine. The server stub then indeed makes a local procedure call on the server.

Fig. 1. Client-Server model.

This model is attractive in many ways. For one thing, the client need not know anything

about the fact that the server is remote. It just makes an ordinary procedure call, with the parameters passed in the usual way (e.g., on the stack). Similarly, the server is called by a local procedure according to the local calling and parameter passing conventions. For another thing, the semantics are straightforward and familiar. Programmers understand the procedure call model much better than the message model.

For all its elegance, however, a number of problems lurk under the surface. Many of these have important implications for the system's reliability. Most of them are directly related to the goal of the remote procedure call—transparency, that is, making it look like a local procedure call.

To begin with, when a program makes a local procedure call, the procedure is executed exactly once, no more and no less. With remote procedure calls, this ideal is unachievable in general. The problem is that the remote server may crash just before or after performing the remote operation, but before sending back the acknowledgement. If the client repeats the request, and it was already carried out, then it will be carried out a second time.

Operations that can be carried out multiple times without harm, such as overwriting a specific disk block are said to be **idempotent**. Unfortunately, most operations that involve communication or I/O are not idempotent. For example, if the request was to a bank server to transfer a large amount of money to a numbered Swiss bank account, one would prefer that operation not be executed by accident a second time.

At first glance you might think that the problem could be solved by having the server record the fact that it was about to perform the operation in a secure way, for example, on stable storage[2]. However, this idea does not work for nonidempotent operations because after recording its intentions the server has to carry out the operation and then send the acknowledgement. In the best case, each of these steps can be done in a single instruction, for example, by setting one bit somewhere. If the server crashes between the two instructions, when it reboots it cannot determine if the crash occurred just before, between, or just after the two instructions.

This observation leads to three classes of remote procedure call systems: those that have ''at least once'' semantics, those that have ''at most once semantics'' and those that have ''don't know'' semantics. In the former class, if the client stub does not get a reply within a specified interval, it just keeps repeating the request until it gets one. The call may be repeated several times, however.

The second kind of semantics is ''at most once.'' One way to implement this is to simply avoid all retransmissions, but then a simple lost message results in a failed execution. A better way is to have the server log all actions *before* performing them, so that if a repeated request comes in, it can be recognized as such and rejected. With this model, the client knows that the call has been performed either 0 or 1 times, but no more.

The third category consists of systems that give no guarantee at all. These have the advantage of being easy to implement.

Transparency also brings other problems with it. Suppose a server is overloaded. A client that does not realize that the lack of response is due to overload may think it is due to lost messages and keep retransmitting, thus making the problem worse.

2.2. Server Crashes

Another source of unreliable behavior is machine failures, either due to hardware or software. These can be split into two categories: server crashes and client crashes. These have different consequences for the system and must be attacked differently. In this section we will look at the problems associated with server crashes and in the next one client crashes.

In general, servers can crash. Obviously one should try to make the servers as reliable as possible, but even perfect software will not act properly if the hardware refuses to work.

Furthermore, making the software perfect is easier said than done. This problem can be approached two ways. One way is to try to get crashed servers back on the air as fast as possible. The other way is to provide multiple servers for redundancy.

Getting crashed servers back up again requires some mechanism to detect when a server has gone done and some way to get it back. Ideally there should also be some mechanism to adjudicate disputes. If the server claims to be up but its clients claim that it is not doing anything, what then? Whatever mechanism is chosen to monitor servers should itself be highly reliable of course.

When the problem of unreliable servers is tackled by having several of each kind, the issue of client-server binding arises. In a system with multiple identical servers (e.g., 3 file servers), at some point a choice must be made about which one a client will use. One can easily imagine a system in which the servers share a common address or mailbox, with each server taking new work out of the mailbox whenever the server is idle. Suppose server 1 takes a request out of the mailbox, carries it out, and then sends an acknowledgement that is subsequently lost.

At this point server 1 crashes. The client times out and retransmits the request, only to have it be taken by server 2 this time, which knows nothing about what server 1 has done recently, because server 1 is currently down and cannot tell it. Server 2 now repeats the request. If the semantics are ‘‘at most once’’ we have a problem. This problem occurs even if server 1 has carefully logged the request and reply in order to filter out repeats.

The difficulty is that the binding between the client and server was automatically broken and reset when the first server went down. Many systems regard automatic rebinding as a step towards fault-tolerant, reliable systems, but we see here that one must be careful.

Another issue related to automatic rebinding of servers is that of state. Some servers may have a long term state that is maintained even after a remote procedure call has terminated successfully. For example, some file servers have an operation OPEN on a file that returns a file descriptor for use in subsequent READs and WRITEs. If multiple instances of such a server exist, problems will arise if the server holding a particular client’s open file table crashes between two remote procedure calls so that subsequent calls go to a new server not having the necessary state.

Of course the system can have a rule that odd-numbered clients always use server 1 and even-numbered clients always use server 2, but such a scheme completely defeats one of the goals of a distributed system, namely, to use redundancy to improve reliability.

Yet another reliability problem associated with binding is authentication. How can the server tell which client sent the message, and how can the client be sure he is sending his data to the real server and not to an imposter? Going through a full authentication protocol, complete with passwords, on every call is not feasible. On the other hand, solutions such as that of Birrell[7] effectively require setting up a long-term encrypted session, thus moving away from the idea of transparency, since now remote procedure calls need to first set up sessions between client and server, but local ones do not.

2.3. Client Crashes

So far we have only looked at the reliability problems caused by server crashes. Client crashes also cause plenty of headaches. When a client starts up a computation on a server and then crashes, the computation continues even though nobody is interested in it any more. Such a computation is called an **orphan**. Having a lot of orphans lying around making random computations does not enhance the reliability of a system. Orphans are most serious when the computation being done by the server takes a substantial amount of time.

Various methods, some fairly draconian, have been proposed for dealing with orphans. One method is to kill off all processes in the whole system every T seconds. This will

certainly kill off all the orphans, but it is something of a nuisance to normal computations.

Another possibility is to have each server periodically check to see if the client that started the current computation is still interested. A variation on this idea is the **dead man's handle**. A client is expected to poll a server working for it periodically. If a poll fails to come in on schedule, the server just kills the computation.

A different approach is to program all clients to log all remote procedure calls on stable storage before making them. When a client reboots after a crash, it checks to see if there were any servers working for it, and if so, tells them to stop. This solution is expensive because writing to disk to log each call doubles the cost of each remote procedure call.

No matter which of these methods is chosen for killing off orphans, there is always the danger than an orphan will be in the middle of a critical section at the instant that it is killed, or that it holds many locks on resources. In this case, killing the orphan can lead to race conditions and deadlocks.

Even if a method can be found to kill off all orphans, it may well be that an orphan has created some long term state that will cause other actions to happen later. For example, a file may have been put in a queue for subsequent processing elsewhere in the system. Thus even after an orphan has been killed off, some other processor may examine the queue, find the work, and start up another orphan.

Let us now briefly look at some systems that have attempted to deal with server and client crashes. Borg et al.[8] have described a system in which each process has a backup process running on a different processor. Whenever a client sends a message to a server, it also sends the same message to the server's backup, as shown in Fig. 2. Similarly, replies are sent to both the client and its backup. The operating system takes care of coordinating and synchronizing all the messages.

Fig. 2. Each process has its own backup.

The idea behind this technique is that if a process crashes, its backup, on another processor, will be available to take over. Of course this scheme requires doubling the number of processors. Powell and Presotto[9] have proposed a simpler scheme that only requires one extra process, instead of doubling the number of processes. In their scheme, shown in Fig. 3, there is a single **recorder process** that logs all messages sent on the network.

Fig. 3. A recorder process logs all message traffic.

If a process crashes, a new processor can be allocated, and the code of the crashed process loaded into it. Then the recorder carefully spoon feeds the new process all the messages

it has saved, in order to get the new process into the same state as the old one was when it went down. Messages sent by the process while it is getting to the point where the old one was are intercepted just before they are sent, to prevent their recipients from being confused. When the new process gets to the point that the old one was, it switches into normal mode, so that messages really are sent.

Processes can also make checkpoints of themselves from time to time if they wish. Doing so means that if a process crashes, the checkpoint can be started up and only messages logged after the checkpoint was made have to be replayed.

Powell and Presotto's technique has the advantage of not requiring any overhead during normal operation. However, it does implicitly presume that all messages are correctly received and logged by the recorder.

A different approach to reliability is Cooper's[10] replicated procedure call. In Cooper's model, each client process is in reality *n* processes running in parallel and executing the same code. Similarly, each server consists of *m* parallel processes. When a client calls a server, each client process sends a message to each server process.

When the replies come back to the client, they are compared. One possible comparison algorithm is to vote. Whichever answer occurs the most times is declared the winner, and given to each client. The clients then continue their work. In this manner, an occasional error is simply voted down, thus giving a degree of fault tolerance.

2.4. Data Integrity

Another key reliability issue is data availability and integrity. If data are frequently inaccessible because some key server is down, users will perceive the system as unreliable. This problem can be dealt with to some extent by having multiple servers of each type, each holding its own private copy of the data. As long as the data are never changed (or very rarely changed), this solution works well. However, if updates are frequent, the redundancy itself introduces problems.

The main problem, of course, is that having multiple copies of the data introduces the possibility of the various copies becoming different over the course of time. Before looking at the replication problem, let us first take a look at the good old days of magnetic tape. In those days, it was common for companies to have a master tape with their current inventory of products. Each day tapes containing the day's purchases and sales would be brought to the computer center. The master tape, an update tape, and a blank tape would be mounted, and a job run making an updated master on the blank tape. Then the next update tape would be run with the new master, and so on.

The nice thing about this system was that if the computer crashed at any instant, it was always possible to go back to the original or any other master tape and start everything again. When magnetic disks were introduced, systems began updating records in place, losing the idempotency of the tape scheme. Furthermore, when multiple update runs were allowed at the same time, sophisticated concurrency control algorithms had to be introduced to make the updates serializable while avoiding deadlock. In this view, the very concept of updating files in place on the disk is seen as a major source of unreliability. When the situation is further complicated by having the work distributed over multiple machines, the potential reliability problems become even worse.

Assuming the problems of concurrency control and serializability on a single machine can be dealt with by conventional means, the issue of replication can be dealt with in several ways. The first way is to have a master copy with multiple backups. This scheme closely resembles the old tape system. After the master copy has been updated, the changes have to be propagated to the backups.

The second way is to update all the copies in parallel, but when inconsistencies arise, to

vote[11-12]. In this way minority viewpoints can be stamped out.

A third scheme is regeneration[13]. When an update is done, the server doing the update arranges for multiple copies to be made. If one of those subsequently becomes disconnected or unavailable, the server just abandons the missing copy and generates a new one.

3. RELIABILITY IN AMOEBA

In this section we will look at the Amoeba distributed operating system[14-17] to see how reliability issues have been dealt with in a real system. First we give a brief introduction to Amoeba.

Amoeba is a distributed operating system that has been designed and implemented at the Vrije Universiteit and the Centrum voor Wiskunde en Informatica. It runs on a collection of 40 Motorola 68000s, 68010s, and 68020s connected by a 10 Mbps local area network. The conceptual model behind the system is the abstract data type. Client processes can perform operations on objects managed by servers. These operations are implemented by having the clients send messages to the servers, with the servers sending the results of the operations back to the clients. This is a simple form of remote procedure call.

Both client and server processes, called *clusters*, can consist of multiple *tasks* that conceptually run in parallel within the same address space. While one task is blocked waiting for a message, another one can be running. Many servers are implemented as a collection of tasks, each of which starts out waiting for a message. When a request to perform an operation arrives, it is given to one of the tasks at random. If that task should later block (e.g., waiting for a disk), another task in the cluster can run on behalf of a different client. Synchronization is achieved by never switching from one task to a different task in the same cluster except when the current task is logically blocked. The scheduler can switch between clusters at will, however.

The Amoeba system consists of four basic components, as shown in Fig. 4. The workstations are used to provide a multi-window interface to the user, as well as some local computing such as editing. The pool processors can be dynamically allocated as needed for compilations, text formatting, or doing any other work. An *n-pass* compiler, for example, can be arranged to allocate, use, and then return *n* pool processors, one per pass.

Fig. 4. An Amoeba system has four components.

The system also contains specialized servers with dedicated functions, such file servers, bank servers, and boot servers. Finally, the fourth component is the gateway to other Amoeba systems. Soon Amoeba will be running at five sites in three countries, all interconnected by a wide-area network.

Identical Amoeba kernels run on all the machines. The kernels are intentionally small, basically handling only communication and low level memory management. Files, process management, and even protection and accounting are all handled at the user level.

Objects are protected by capabilities, as shown in Fig. 5. Each capability contains a *port* field that is used to identify the server or client being addressed and an object field, used to identify the specific object to be manipulated. Object numbers are analogous to i-node numbers in UNIX.† Next comes a rights field, telling which operations the holder of the capability may perform on the object. Finally, there is a random number that prevents users from forging capabilities. Capabilities are directly handled by user processes, outside the kernel.

Fig. 5. An Amoeba capability.

The random number field is crucial to the protection scheme, hence to the reliability of the system. When an object is created, the creating server allocates an ‘‘i-node’’ for it and puts a random number in it. It then EXCLUSIVE ORs the rights bits (initially all 1s) with the random number and runs the result through a one-way function[18]. used for all objects. The output of the one way function is put into the random field of the capability. The rights bits are included in the capability in plaintext.

When a client performs an operation on an object, the capability for the object is sent to the server to identify the object. The server then uses the object number contained in the capability as an index into its tables to find the random number. The random number thus found is EXCLUSIVE ORed with the plaintext rights field and run through the one-way function. If the output is the same as the capability’s random number, the capability (including the plaintext rights bits) is accepted as valid. This protection system and several variations on it are described in more detail in Tanenbaum et al.[17].

3.1. Interprocess Communication

The form of remote procedure call used by Amoeba has ‘‘at most once’’ semantics. For most applications this is preferable to ‘‘at least once’’ and certainly better than ‘‘don’t know.’’ We will now describe how these semantics are implemented.

When a remote procedure call is made, the client calls a stub procedure that locates a server based on the port number present in the capability belonging to the object to be operated upon. The location is done by first looking in a cache. If that fails, a broadcast is done. If multiple servers handle the object class in question, the stub selects one of them, and gets its process identifier (pid).

Then a message is sent to the selected server process. Normally, the server will perform the operation and send back a reply. If the server’s reply is not forthcoming within a certain time interval, the server’s stub times out and acknowledges receipt of the request so the client will know that it arrived safely and that the server is hard at work on it. When the server’s reply finally gets back to the client, the client’s stub sends an acknowledgement back to the server, which terminates the call.

If it has received an acknowledgement but no reply to the request itself, at a certain point the client gets nervous and sends an ‘‘Are you alive?’’ query to the server, which is answered immediately. On the other hand, if the client has heard nothing at all from the server, not even the acknowledgement of the request, it eventually times out and retransmits

† UNIX is a Registered Trademark of AT&T Bell Laboratories.

the request. When the server sees the retransmitted request, which bears the same source and request number as the original, it can recognize the request as a retransmission and just send the reply again or at least just acknowledge receipt of the request if the result is not yet available.

Now consider what happens if the server crashes. The client stub eventually detects that the server process is down when it fails to get answers to its ‘‘Are you alive?’’ messages. If the client stub has enough knowledge of the specific operation to be sure that it is idempotent, it can locate another server and repeat the operation. In this case it does not matter that the operation was executed more than once.

On the other hand, if the stub does not know whether or not the operation is idempotent, it simply reports back failure to the client, meaning that that the operation has been performed either 0 or 1 times, but not more.

3.2. Server Crashes

The communication mechanism is not the only part of Amoeba that was designed with reliability in mind. There is also a *boot server* whose job is to make sure that processes (typically servers) that are supposed to be alive are in fact alive. It does this by periodically probing the registered servers to see if they are still functioning.

All the long-lived servers, such as the file servers, normally register with the boot server when the system comes up. This registration consists of providing the boot server with the message to be sent to the server and the reply that the server is supposed to send back, the frequency at which these probes are to take place, the number of probes to make before declaring the server dead, and the procedure for creating a new server to replace one that has crashed.

The procedure used to reincarnate a crashed server depends on the nature of the crash. If the server is dead but the kernel on its machine is still working, then the boot server instructs the kernel to create a new server process to replace the old one.

If the entire machine has crashed, then the boot server sends a special packet on the network that is detected by the interface card, and which results in the interface asserting a RESET signal on the crashed machine’s bus. This signal causes the machine to reboot itself by jumping to a program in a ROM. The ROM program and the boot server together download a new kernel into the machine, at which time the server can be restarted. If the machine cannot be started up at all, the boot server gets another processor and starts the server there. This whole procedure is fully automated; it happens without human intervention.

The only other issue concerning the boot server is the reliability of the boot server itself. Multiple copies of the boot server run, each one communicating with all the other ones. If one of the boot servers crashes, the remaining ones regenerate it using the procedure just described.

3.3. Client Crashes

Orphans are prevented in Amoeba by using the ‘‘Are you alive’’ messages as a dead man’s handle. If a server is making a long computation, it expects to get ‘‘Are you alive messages’’ periodically. If these messages cease to arrive, the server concludes that the client is dead and kills the orphan itself.

Although the orphan detection mechanism is useful for ridding the system of unwanted computations, in many circumstances it is desirable that clients be fully fault tolerant, meaning that a client, especially one running in parallel on multiple pool processors, itself notices crashes of some of its processors and recovers from them in a transparent way. Several applications have been programmed in this way. Below we will briefly sketch two of these, the traveling salesman problem and parallel alpha-beta search.

The traveling salesman problem consists of finding the shortest route that a salesman can use to visit all the cities in his territory exactly once. Roughly speaking, the Amoeba approach is to have a procedure, *traverse*, that takes as input a partial path, the set of cities as yet unvisited, and the length of the best total path found so far[19]. This procedure forks off a process for each unvisited city to investigate all paths with that city as the next step. Each process simply runs *traverse*, with a partial path one city longer and the set of unvisited cities one smaller. The recursive forking of parallel processes continues until a certain depth in the tree has been attained, at which point the residual tree is searched completely by one process. Variations of this search strategy have also been tried.

The reliability comes from the fact that if a process fails to report back its findings within a certain time, and also fails to respond to the ''Are you alive'' messages, the process that invoked it just asks for another pool processor and starts the work all over again. Higher levels in the tree do not even know that a fault has been detected and corrected. In this way the program will be executed correctly even in the face of repeated multiple processor crashes.

The other reliable application that has been tested is heuristic search for the game reversi (Othello) using the alpha-beta algorithm. At each board position a process is generated for each legal move. Although the details of alpha-beta make this application somewhat different than the branch and bound algorithm used for the traveling salesman, again if a process crashes, its parent just finds someone else to do the work. As we mentioned in the introduction, the fact that the parallelism is visible to the application makes it possible to exploit it for better reliability.

3.4. Data Integrity

File servers in Amoeba are user-level processes, so there can be several of them running at once, providing different services and serving different clients. Some of the file servers have been designed to provide UNIX-file service, others have been designed for high performance, but there is also one whose goal is high reliability. This one, called FUSS (Free University Storage System) is described by Tanenbaum and Mullender[15] and is sketched below.

The technique used by FUSS to provide high reliability is the *immutable file*. When a process wants to update a file, it asks FUSS to create a new version of the file and return a capability for the copy. (Actually the file is not copied. Shadow pages are used, but this is really just an optimization.) The process can then modify the copy as it wishes. When it is done, the process tells FUSS to *commit* the file, making the copy the new file. Thus a file is really a sequence of versions, none of which is ever modified once it has been committed. Modifying a file consists of atomically replacing a file with a new version.

This design is more reliable than the traditional update-in-place file system because updating a file consists of preparing the new file and then at the last minute switching one pointer. If the file server crashes, either the old file or the new file will be present when it comes up again, but never a mixture of the two. By appropriate logging of intentions on a disk, the server can be made to eventually complete the update no matter how often it crashes. The atomic update property is especially important if two or more processes are simultaneously updating the same file. FUSS offers a choice between locking and optimistic currency control, but in both cases, an update to a file (or even a set of files) is atomic.

Work is currently in progress to extend these ideas to general objects. The idea is that any object should be representable as a sequence of versions, with the update from the old version to the new one being done atomically. This can be achieved by having a directory server that maps ASCII strings onto capabilities, or more generally, onto sets of capabilities. In effect, a directory is an unordered collection of lines, each containing a ASCII object name

followed by set of capabilities. The capabilities are for replicas of the same object.

A directory is thus simply a mapping of ASCII names onto sets of objects. A directory is itself an object, so directories can contain capabilities for other directories, giving rise to a directory hierarchy, or even a general graph, if that is desired.

The principal operation on a directory object is to present the directory server with a capability for a directory and an ASCII string to be looked up in that directory. The server then looks up the given string in the directory and returns the full set of capabilities that correspond to that string, if any. The client can then choose one of them at random to use. If that one is not available, it can choose another one.

The idea of having the directory entry contain multiple capabilities has been done to enhance the reliability. Because files (and objects generally) are immutable, once a new version of an object has been created, the directory server can arrange for backup copies of the object to be made at its leisure (lazy backup). There is no problem with race conditions because the object cannot change. The worst that can happen is that the version being backed up becomes obsolete before all the backups have been created, in which case some extra work may have been done for nothing, but the file system integrity is never affected.

Updating a directory entry is done by sending the directory server a capability for a directory, an ASCII string, the capability for the object being replaced, the capability for the new object, and a count specifying how many backup copies should be made and maintained. The directory entry is updated atomically—either it happens or it fails, but there is never half an update. Notice that the replication effort is managed by the directory server, so it need not be duplicated in each object server. This is possible because objects are immutable. Once an object has been committed, it never changes; it can only be replaced in its entirety by a new object.

The update operation requires the old capability as a parameter so the directory server can verify that the object being replaced is still the current object. If the old capability is not present in the set of capabilities for the given string, the directory server can see that another update has transpired in the meantime, so the update operation fails. This scheme is a form of optimistic concurrency control. Put in other terms, if two clients each look up a given string in a given directory, and then both try updating the corresponding object, only the first update will succeed. Objects can also be locked, to allow a more conventional update strategy.

3.5. Other Reliability Features of Amoeba

Another area that affects system reliability is resource management. If one user or process consumes too many resources, the rest of the users and processes will suffer the consequences. For this reason Amoeba has a *bank server* that can be used as a general tool for resource management.

The bank server manages bank accounts in various currencies. As an example of its use, consider a file server that wished to implement a quota system to give each user at most 1000 disk blocks. Each user would be given a bank account containing, say, 1000 zlotys, each good for one disk block. Every time a user wanted another disk block, he would first have to transfer 1 zloty to the file server's account to pay for it in advance. When the block was freed, the user would get his zloty back.

Other currencies can be used for other resources. CPU time could be charged in yen, phototypesetter pages in guilders, etc. The *policies* (e.g., who gets how much money, whether currencies are convertible) are decided by the servers, but the basic *mechanism* (managing the accounts, logging transactions, transferring money between accounts atomically, maintaining caches for efficiency, etc.) is done by the bank server, so that each individual server need not run its own administration.

Try as we may to build a reliable system, there are going to be bugs in it. For this reason, Amoeba has been designed in such a way to be able to catch faults and handle them. To see how this mechanism works, we have to take a look at how processes are managed in Amoeba. When a user types a command to the shell, the shell creates a *mother* process to oversee the execution of the command. The mother process allocates a processor from the processor pool, asks the Amoeba kernel on that machine to allocate sufficient memory for the new process, and then downloads the program to be executed to the processor for execution.

Normally, the mother process does not intervene in the execution of the program on the pool processor. It simply waits until the program terminates to clean it up and report back its status. However, it is possible to tell the pool processor's kernel to catch all system calls and other kernel traps, and send them to the mother process for processing.

In this way, for example, it is possible to take a *binary* program compiled to run on 68000 UNIX (i.e., not on Amoeba) and run it on a pool processor, even though the Amoeba kernel knows nothing at all about UNIX. The UNIX system calls are effectively all passed to the mother process for execution. If the mother process happens to be running on a 68000 UNIX system (which is easy to arrange), it can just execute the system calls locally and send back the results.

This same mechanism is used for debugging. When a process on a pool processor gets a memory fault, illegal instruction, or other kernel trap, the pool processor's kernel does a remote procedure call with the mother process telling it what happened. The mother process contains a debugger that can print a message on the user's terminal and then wait for input instructing it what to do. There are commands to examine and print memory and so on. These are handled by messages between the mother process and the kernel on the pool processor.

3.6. Summary

Reliability considerations have influenced the Amoeba design in a number of ways. These include the scheme for protecting objects with cryptographically secure capabilities, the communication mechanism with ''at most once'' semantics and orphan extermination, the boot server for automatically rebooting dead processes, the file server with immutable files, the directory sever with atomic update on replicated objects, the bank server for limiting resource usage, and the hooks for debugging. In addition, Amoeba has been used for explicitly programming fault tolerant applications such as the traveling salesman and heuristic search.

4. REFERENCES

[1] Tanenbaum, A.S., and van Renesse, R.: ''Distributed Operating Systems,'' *Computing Surveys*, vol. 17, pp. 419-470, Dec, 1985

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

[3] Zimmermann, H. ''OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection,'' *IEEE Trans. Commun.*, vol. COM-28, pp. 425-432, April 1980.

[4] Birrell, A.D., and Nelson, B.J. ''Implementing Remote Procedure Calls,'' *ACM Trans. Comput. Systems*, vol. 2, pp. 39-59, Feb. 1984.

[5] Nelson, B.J. ''Remote Procedure Call,'' Tech. Rep. CSL-81-9, Xerox PARC, 1981.

[6] Spector, A.Z. ‘‘Performing Remote Operations Efficiently on a Local Computer Network,’’ *Commun. ACM*, vol. 25, pp. 246-260, April 1982.

[7] Birrell, A.D. ‘‘Secure Communication Using Remote Procedure Calls,’’ *ACM Trans. Comput. Syst.*, vol. 3, pp. 1-14, Feb. 1985.

[8] Borg, A., Baumbach, J., and Glazer, S. ‘‘A Message System Supporting Fault Tolerance,’’ *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 90-99, 1983.

[9] Powell, M.L., and Presotto, D.L. ‘‘Publishing-A Reliable Broadcast Communication Mechanism,’’ *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 100-109, 1983.

[10] Cooper, E. ‘‘Replicated Distributed Programs,’’ *Proc. Tenth Symp. on Oper. Syst. Prin.*, ACM, 1985.

[11] Thomas, R.H.: ‘‘A Majority Consensus Approach to Concurrency Control,’’ *ACM Trans. on Database Syst*, vol 4, pp. 180-209, June 1979.

[12] Gifford, D.K.: ‘‘Weighted Voting for Replicated Data,’’ *Proc. Seventh Symp. on Operating Syst, Prin*, ACM, 1979.

[13] Pu, C., Noe, J.D., and Proudfoot, A. ‘‘Regeneration of Replicated Objects: A Technique and its Eden Implementation,’’ *Proc. Second Int’l Conf. on Data Engineering*, pp. 175-187, Feb. 1986.

[14] Mullender, S.J., and Tanenbaum, A.S. ‘‘Protection and Resource Control in Distributed Operating Systems,’’ *Computer Networks*, vol 8, pp. 421-432, Nov. 1984.

[15] Mullender, S.J., and Tanenbaum, A.S. ‘‘A Distributed File Service Based on Optimistic Concurrency Control,’’ *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 51-62, 1985.

[16] Mullender, S.J., and Tanenbaum, A.S. ‘‘The Design of a Capability-Based Distributed Operating System,’’ *Computer Journal*, vol. 29, pp. 289-299, Aug. 1986.

[17] Tanenbaum, A.S., Mullender, S.J., and van Renesse, R.: ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ *Proc. 6th Int’l Conf, on Distr. Comp. Syst.*, IEEE, pp. 558-563, 1986.

[18] Evans, A., Kantrowitz, W., and Weiss, E.: ‘‘A User Authentication Scheme not Requiring Secrecy in the Computer,’’ *Communications of the ACM*, vol. 17, pp. 437-442, Aug. 1974.

[19] Bal, H.E., van Renesse, R., and Tanenbaum, A.S. ‘‘A Distributed, Parallel, Fault Tolerant Computing System,’’ Report IR-106, Dept. of Math. and Comp. Sci., Vrije Univ., Oct. 1985.