CERN 93–03
6 July 1993

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

1992 CERN SCHOOL OF COMPUTING

Scuola Superiore G. Reiss Romoli, L'Aquila, Italy
30 August–12 September 1992

PROCEEDINGS
Editor: C. Verkerk

GENEVA
1993

Introduction to Distributed Systems*

Sape J. Mullender

Based on a Lecture by Michael D. Schroeder

The first four decades of computer technology are each characterized by a different approach to the way computers were used. In the 1950s, programmers would reserve time on the computer and have the computer all to themselves while they were using it. In the 1960s, batch processing came about. People would submit their jobs which were queued for processing. They would be run one at a time and the owners would pick up their output later. Time-sharing became the way people used computers in the 1970s so that users could share a computer under the illusion that they had it to themselves. The 1980s are the decade of personal computing: people have their own dedicated machine on their desks.

The evolution that took place in operating systems has been made possible by a combination of economic considerations and technological developments. Batch systems could be developed because computer memories became large enough to hold an operating system as well as an application program; they were developed because they made it possible to make better use of expensive computer cycles. Time-sharing systems were desirable because they allow programmers to be more productive. They could be developed because computer cycles became cheaper and computers more powerful. Very large scale integration and the advent of local networks has made workstations an affordable alternative to time sharing, with the same guaranteed computer capacity at all times.

Today, a processor of sufficient power to serve most needs of a single person costs less than one tenth of a processor powerful enough to serve ten. Time sharing, in fact, is no longer always a satisfactory way to use a computer system: the arrival of bit-mapped displays with graphical interfaces demands instant visual feedback from the graphics subsystem, feedback which can only be provided by a dedicated, thus personal, processor.

The workstations of the 1990s will be even more powerful than those today.

---

*. The text of this paper was taken from *Distributed Systems*, S.J. Mullender (ed.), ACM Press, 1989, ISBN 0-201-41660-3.

We foresee that a typical workstation will have a high-resolution colour display, and voice and video input and output devices. Network interfaces will allow communication at rates matching the requirements of several channels of real-time video transmission.

A time-sharing system provides the users with a single, shared environment which allows resources, such as printers, storage space, software and data to be shared. To provide users access to the services, workstations are often connected by a network and workstation operating-system software allows copying files and remote login over the network from one workstation to another. Users must know the difference between local and remote objects, and they must know at what machine remote objects are held. In large systems with many workstations this can become a serious problem.

The problem of system management is an enormous one. In the time-sharing days, the operators could back up the file system every night; the system administrators could allocate the available processor cycles where they were most needed, and the systems programmers could simply install new or improved software. In the workstation environment, however, each user must be an operator, a system administrator and a systems programmer. In a building with a hundred autonomous workstations, the operators can no longer go round making backups, and the systems programmers can no longer install new software by simply putting it on the file system.

Some solutions to these problems have been attempted, but none of the current solutions are as satisfactory as the shared environment of a time-sharing system. For example, a common and popular approach consists of the addition of network-copy commands with which files can be transferred from one workstation to another, or — as a slightly better alternative — of a *network file system*, which allows some real sharing of files. In all but very few solutions, however, the user is aware of the difference between local and remote operations. The real problem is that the traditional operating systems which still form the basis of today's workstation software were never designed for an environment with many processors and many file systems. For such environments a *distributed operating system* is required.

The 1990s will be the decade of distributed systems. In distributed systems, the user makes no distinction between local and remote operations. Programs do not necessarily execute on the workstation where the command to run them was given. There is one file system, shared by all the users. Peripherals can be shared. Processors can be allocated dynamically where the resource is needed most.

A distributed system is a system with many processing elements and many storage devices, connected together by a network. Potentially, this makes a distributed

system more powerful than a conventional, centralized one in two ways. First, it can be more reliable, because every function is replicated several times. When one processor fails, another can take over the work. Each file can be stored on several disks, so a disk crash does not destroy any information. Second, a distributed system can do more work in the same amount of time, because many computations can be carried out in parallel.

These two properties, *fault tolerance* and *parallelism* give a distributed system the potential to be much more powerful than a traditional operating system. In distributed systems projects around the world, researchers attempt to build systems that are fault tolerant and exploit parallelism.

1 Symptoms of a distributed system

The fundamental properties of a distributed system are *fault tolerance* and the possibility to use *parallelism*. What does a system have to have in order to be 'distributed'? How does one recognize a distributed system?

Definitions are hard to give. Lamport was once heard to say: 'A distributed system is one that stops you from getting any work done when a machine you've never even heard of crashes.' More seriously, Tanenbaum and van Renesse [1985] give the following definition:

> A *distributed* operating system is one that looks to its users like an ordinary centralized operating system, but runs on multiple, independent CPUs. The key concept here is *transparency*, in other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a 'virtual uniprocessor', not as a collection of distinct machines.

According to this definition, a *multiprocessor operating system*, such as versions of Unix for Encore or Sequent multiprocessors would be distributed operating systems. Even dual-processor configurations of the IBM 360, or CDC number crunchers of many years ago would satisfy the definition, since one cannot tell whether a program runs on the master or slave processor.

Tanenbaum and van Renesse's definition gives a necessary condition for a distributed operating system, but I believe it is not a sufficient one. A distributed operating system also must not have any *single points of failure* — no single part failing should should bring the whole system down.

This is not an easy condition to fulfill in practice. Just for starters, it means a distributed system should have many power supplies; if it had only one, and it

failed, the whole system would stop. If you count a fire in the computer room as a failure, it should not even be in one physical place, but it should be geographically distributed.

But one can carry failure transparency too far. If the power fails in our building, I can tolerate my system failing, provided it fails cleanly and I don't lose the work I have just been doing — one can't work very well in the dark anyway.

It is dangerous to attempt an exact definition of a distributed system. Instead, Schroeder gave a list of *symptoms* of a distributed system. If your system has all of the symptoms listed below, it is probably a distributed system. If it does not exhibit one or more of the symptoms, it probably isn't one.

A distributed system has to be capable of continuing in the face of single-point failures and of parallel execution, so it must have

- *Multiple processing elements*, that can run independently. Therefore, each processing element, or *node* must contain at least a CPU and memory.

There has to be communication between the processing elements, so a distributed system must have

- *Interconnection hardware* which allows processes running in parallel to communicate and synchronize.

A distributed system cannot be fault tolerant if all nodes always fail simultaneously. The system must be structured in such a way that

- *Processing elements fail independently.* In practice, this implies that the interconnections are unreliable as well. When a node fails, it is likely that messages will be lost.

Finally, in order to recover from failures, it is necessary that the nodes keep

- *Shared state* for the distributed system. If this were not done, a node failure would cause some part of the system's state to be lost.

To see more clearly what constitutes a distributed system, we shall look at some examples of systems.

Multiprocessor computer with shared memory

A shared-memory multiprocessor has several of the characteristics of a distributed system. It has multiple processing elements, and an interconnect via shared memory, interprocessor interrupt mechanisms and a memory bus. The communication between processing elements is reliable, but this does not in itself mean that a multiprocessor cannot be considered as a distributed system. What disqualifies multiprocessors is that there is no independent failure: when one processor crashes, the whole system stops working. However, it may well be that manufacturers, inspired by distributed systems research, will design multiprocessors that are capable

of coping with partial failure; to my knowledge, only Tandem manufactures such machines, currently.

Ethernet with packet-filtering bridges

A bridge is a processor with local memory that can send and receive packets on two Ethernet segments. The bridges are interconnected via these segments and they share state which is necessary for routing packets over the internet formed by the bridges and the cable segments. When a bridge fails, or when one is added to or removed from the network, the other bridges detect this and modify their routing tables to take the changed circumstances into account. Therefore, an Ethernet with packet-filtering bridges *can* be viewed as a distributed system.

Diskless workstations with NFS[†] file servers

Each workstation and file server has a processor and memory, and a network interconnects the machines. Workstations and servers fail independently: when a workstation crashes, the other workstations and the file servers continue to work. When a file server crashes, its client workstations do not crash (although client processes may hang until the server comes back up). But there is no shared state: when a server crashes, the information in it is inaccessible until the server comes back up; and when a client crashes, all of its internal state is lost. A network of diskless workstations using NFS file servers, therefore, is *not* a distributed system.

2 Why build distributed systems?

People are distributed, information is distributed

Distributed systems often evolve from networks of workstations. The owners of the workstations connect their systems together because of a desire to communicate and to share informations and resources.

Information generated in one place is often needed in another. This book illustrates the point: it was written by a number of authors in different countries who needed to interact by reading each other's material and discussing issues in order to achieve some coherence.

---

†. NFS stands for SUN Microsystems' Network File System.

Performance/cost

The cost of a computer depends on its performance in terms of processor speed and the amount of memory it has. The costs of both processors and of memories are generally going down. Each year, the same money, buys a more powerful workstation. That is, in real terms, computers are getting cheaper and cheaper.

The cost of communication depends on the bandwidth of the communication channel and the length of the channel. Bandwidth increases, but not beyond the limits set by the cables and interfaces used. Wide area network cables have to be used for decades, because exchanging them is extremely expensive. Communication costs, therefore, are going down much less rapidly than computer costs.

As computers become more powerful, demands on the man-machine bandwidth go up. Five or ten years ago, most computer users had a terminal on their desk, capable of displaying 24 lines of text containing 80 characters. The communication speed between computer and terminal was 1000 characters per second at most. Today, we consider a bit-mapped display as normal, even a colour display is hardly luxury any more. The communication speed between computer and screen has gone up a few orders of magnitude, especially in graphical applications. Soon, voice and animation will be used on workstations, increasing the man-machine bandwidth even more.

Man-machine interfaces are also becoming more interactive. Users want instant visual (or audible) feedback from their user interface, and the latency caused by distances of more than a few kilometres of network is often too high already.

These effects make distributed systems not only economic, but necessary. No centralized computer could give the required number of cycles to its customers and the cost of the network technology that gets the required number of bits per second out to the user's screen is prohibitive.

Modularity

In a distributed system, interfaces between parts of the system have to be much more carefully designed than in a centralized system. As a consequence, distributed systems must be built in a much more modular fashion than centralized systems. One of the things that one typically does in a distributed system is to run important services on their own machines. The interface between modules are usually remote procedure call interfaces, which automatically impose a certain standard for inter-module interfaces.

Expandability

Distributed systems are capable of incremental growth. To increase the storage or processing capacity of a distributed system, one can add file servers or processors one at a time.

Availability

Since distributed systems replicate data and have built-in redundancy in all resources that can fail, distributed systems have the potential to be available even when arbitrary (single-point) failures occur.

Scalability

The capacity of any centralized components of a system imposes a limit for the system's maximum size. Ideally, distributed systems have no centralized components, so that this restriction on the maximum size the system can grow to does not exist. Naturally, there can be many other factors that restrict a system's scalability, but distributed system designers do their best to choose algorithms that scale to very large numbers of components.

Reliability

Availability is but one aspect of reliability. A reliable system must not only be available, but it must do what it claims to do correctly, even when failures occur. The algorithms used in a distributed system must not behave correctly only when the underlying virtual machine functions correctly, they must also be capable of recovering from failures in the underlying virtual machine environment.

3 The complexity of distributed systems

The main reason that the design of distributed systems is so hard is that the enormous complexity of these systems is still well beyond our understanding. However, we now understand many aspects of distributed systems and many sources of complexity. We also know how to deal with quite a few of these sources of complexity, but we are still a long way from really understanding how to design and build reliable distributed systems. What causes the extraordinary complexity of distributed systems?

One can answer this question to some extent by comparing distributed computer systems to the railway system, another 'distributed system' that most people are

familiar with. The railway system as we know it today has taken a century and a half to develop into a safe and reliable transport service. It took time before the complexity of the railway system was understood. The development of its safety has been at the cost of errors and mistakes that have cost many lives. It is often the case that, only after the fact, it is discovered that an accident was caused by a perfectly obvious design oversight — yet nobody had seen it beforehand.

Distributed computer systems have only been around for a decade or so, but they are every bit as complicated to design as national railway networks and it will take many generations of distributed systems before we can hope to understand how to build one properly. Just as the railway system has become safe only at the cost of many accidents, so distributed systems only become reliable and fault-tolerant at the cost of many system crashes, discovering design bugs and learning about system behaviour by trial and error.

The basic source of the complexity of distributed systems is that an interconnection of well-understood components can generate new problems not apparent in the components themselves. These problems then produce complexity beyond our limits of methodical understanding. Much of this complexity is apparent though the unexpected behaviour of systems that we believe we understand.

As an illustration of unexpected behaviour consider the well-known fairness of the token ring network (Saltzer and Pogran [1980]). In this network, a station may obtain the token to send one packet at a time; then the other stations get an opportunity to send a packet if they have one. It would appear that station A sending a large, multi-packet message to a server S cannot lock out B from sending a message simultaneously. It turned out the unexpected behaviour had nothing to do with the principle of the token ring, but with the design of the token-ring interface board. Sventek et al. [1983] discovered that the receive circuitry of the interface board misses a packet when it immediately follows another one. This leads to the following scenario.

1. A sends the first packet of a message to S, which is received successfully.
2. B sends the first packet of a message to S, immediately behind A's packet, but it is ignored because it is too close behind the previous packet.
3. A sends the second packet immediately behind B's first packet. Again, A's packet is received, because S has recovered from A's first packet by now.
4. B sends the second packet or retransmits the first (token rings often have a *packet-seen* bit which tells the sender whether the receiving interface has accepted the packet). In either case, S ignores the packet again because it immediately follows A's second packet.

This goes on, of course, and the effect is that B is totally locked out from

sending because the intended fairness of the token ring actually guarantees that A and B send alternate packets; A's packets are always received and B's packets are never received.

The above example nicely illustrates the sort of problems to which combinations of well-understood components can lead. In this case, the token protocol was well understood and the receive interface too, but the combination of the two caused an effect that wasn't foreseen by the designers.

In some cases, formal methods can be used to help predict what will happen when two systems are interconnected. However, these methods are only of limited help, especially when we don't understand the systems or the interconnection incompletely, or when we do not have the tools to describe them formally.

Complexity limits what we can build. Schroeder calls the problems caused by this complexity *systems problems*. Some examples of systems problems are:

Interconnection. A very large number of systems problems come about when components that previously operated independently are interconnected. This has been very visible when various computer networks for electronic mail were interconnected (the Unix mail system, the Bitnet mail system, X.400, to name but a few). Interconnection problems also had to be solved when it became desirable to connect the file systems of different Unix machines.

Interference. Two components in a system, each with reasonable behaviour when viewed in isolation, may exhibit unwanted behaviour when combined. The problem in the token-ring example above is an example of unexpected interference of the properties of the token-passing protocol and the receive circuitry of the network interfaces.

Propagation of effect. Failures in one component can bring down a whole network when system designers aren't careful enough. Two examples from the ARPANET can serve to illustrate this. A malfunctioning IMP (a message-processing node of the network) announced to its neighbours that it had zero delay to every other node in the network. Within minutes, large numbers of nodes started sending all their traffic to the malfunctioning node, bringing the network down. Another time, an east-coast node, due to an error, used the network address of a west-coast node, causing major upheaval in the network. Obviously, design effort is required to localize the effect of failures as much as possible.

Effects of scale. A system that works well with ten nodes may fail miserably when it grows to a hundred nodes. This is usually caused by some resource that doesn't scale up with the rest of the system and becomes a bottleneck, or by the use of algorithms that do not scale up. The Grapevine system, developed

at Xerox PARC (Birrell et al. [1982]), is an excellent example of a system that grew to the limits imposed by its design, and perhaps a little beyond. This is described in Birrell et al. [1984].

Partial failure. The fundamental difference between traditional, centralized systems and distributed systems is that in a distributed system a component may fail, but the rest of the system continues to work. In order to exploit the potential fault tolerance of a distributed system, it is necessary that a distributed application is prepared to deal with partial failures. Needless to say, this is a considerable source of additional complexity in the design of fault-tolerant applications.

To some extent, all of these problems exist in all computer systems, but they are much more apparent in distributed systems: there are just more pieces, hence more interference, more interconnections, more opportunities for propagation of effect, and more kinds of partial failure.

3.1 Functional requirements as a source of complexity

Distributed systems are complex because what they have to do is complex. To illustrate this, we return to our railway example: Just outside Mike Schroeder's office in Palo Alto, California is the Palo Alto station on the San Francisco to San Jose line, which caters to commuter traffic in the Bay Area. The schedule for these trains is a straightforward one to design: the trains run at a frequency that depends primarily on the average number of travellers at each time of day.

When I go to work in the morning, I take the train from Amsterdam to Enschede, which is where my university is. This railway line forms part of a dense national railway network with hundreds of stations on dozens of lines. Here, the train schedules are complicated by a few orders of magnitude by the requirement that passengers have to change trains to reach their destination and that they usually expect that connecting trains are synchronized to within a few minutes.

The Dutch railway system has more demanding functional requirements (people travel in a two-dimensional mesh network rather than on a straight line) and therefore a more complex train schedule. The difference in complexity is enormous; it increases much more rapidly than linear with the number of lines.

As another example, consider three file systems, one for a stand-alone personal computer, one for a time-sharing system and one for a highly-available distributed system.

Designing the PC file system involves a certain amount of complexity, of course, but designing the time-sharing system file system involves a large amount of additional complexity. Here, there are issues of authentication, access control, main-

taining quota, concurrency control for read/write sharing, and many others.

The distributed file system has to be more complex again so that it can meet the additional requirements of a highly available fault-tolerant file system: mechanisms for file location, coordination of replicated server state, recovery mechanisms from partitions or partial failure, and so forth.

Again, a longer list of functional requirements causes a large increase in complexity of the resulting system. The complexity of distributed systems stems to a large extent from the requirements of fault-tolerance, availability, scalability and performance which we impose.

3.2 Economic necessity as a source of complexity

Another source of complexity is that the simple solution cannot always be used: sometimes it is too expensive. Again, the railway serves to illustrate the notion of complexity caused by economic necessity. A single-track railroad through the mountains has a high uni-directional capacity — one can probably run a train every fifteen minutes — but a very long latency to reverse flow. Before a train can go in the other direction, the track must be empty. If the railroad is a long one, this can take many hours.

There are two ways to reduce the time needed to reverse traffic and increase the bi-directional capacity. The simple one is to double the tracks from beginning to end. Traffic in one direction is then independent of the traffic in the other direction. In fact, the operation of the line becomes even simpler than before: reversing traffic does not require co-ordination between the end points in order to know how many trains are still on the way.

Laying double tracks along the length of a mountain line is extremely expensive, so the most economic solution is to build a number of passing sidings in the track. This complicates the operation of the line considerably, however. The trains have to be carefully scheduled, signals are needed to tell the train drivers when they can go, there are limits on the length of trains, there is a risk of deadlock and there is a higher risk of collision.

Examples of complexity for reasons of economy abound in computer systems as well. Most time-sharing computer systems have paged virtual memory, because real memory for peak demand is too expensive; real memory for the working sets of active programs is affordable.

Long-haul networks are usually mesh networks, because fully interconnected networks are very much more expensive. Again the price is increased complexity: routing algorithms have to be designed, buffering is necessary to manage merging traffic, and flow-control mechanisms must prevent traffic between two nodes from

blocking traffic between other nodes.

4 Useful techniques for building a distributed system

This chapter ends with a list of techniques that are important in building distributed systems. The list is by no means complete — we have probably missed some essential ones — but it is a useful list of techniques that may be applied where appropriate.

Replicate to increase availability

Once it is stated in black and white, *'replicate to increase availability'*, it becomes obvious. A single copy of something is not available when the machine it is on is down. The only way to make a service, a data item, or a network connection available across machine crashes and network failures is to replicate it.

Replicating non-changing data or stateless services is trivial. The data item can be copied any number of times, and one can bring up as many servers as one pleases. Since the data item is immutable, and the service is stateless, no information needs to be exchanged between the replicas when they are accessed.

Unfortunately, data often does change, and nearly all useful services have state. Even so-called stateless file servers (SUN's NFS is an example) have state: the contents of the files. The statelessness refers to protocol state, not file state. When one replica changes without the others, they become *inconsistent*. Accessing one replica is no longer the same as accessing another. This may or may not be a problem, but it certainly needs a system designer's attention.

Guaranteeing consistency while at the same time maintaining a large degree of availability is a very difficult problem. The most trivial algorithm for maintaining consistency is to lock every replica before an update, to update every replica and to unlock each one again, but obviously, this makes availability worse than that of a single copy — every replica must be working in order to make progress. Part VI is devoted to replication and the consistency versus availability problem is addressed there.

Trade off availability against consistency

There is a trade off between availability and consistency. In some cases, it is all right to sacrifice some availability to achieve absolute consistency — one often has to, anyway — while in other cases some (controlled) inconsistencies can be allowed to achieve better availability.

An example of a service where some consistency may be sacrificed for better availability is the electronic equivalent of the phone book: a network name service. Phone books are out of date by the time they land on the doormat, but people can live with them quite comfortably — if the number in the phone book turns out to be invalid, one can always call directory information. The same applies to the network name service. It usually maps things like mailboxes to machine addresses, and, if someone moves from one machine to another, the change of address does not reach all replicas of the directory simultaneously. But this doesn't matter: when an old address is used, there is usually a forwarding address to the correct site, and if there isn't, other, more up-to-date versions of the directory service can be consulted, or an enquiry can even be broadcast to all name servers.

Cache hints if possible

One of the most useful examples of a trade off between availability and consistency is a cache of hints. A hint is the saved result of some operation that is stored away so that carrying out the operation again can be avoided by using the hint. There are two additional important things to note about hints:

1. A hint may be wrong (obsolete), and
2. When a wrong hint is used, it will be found out in time (and the hint can be corrected).

Since the operation that was saved as a hint can always be redone, hints can safely be cached. When a hint is dropped from the cache, the operation will just have to be redone to reproduce the hint.

The telephone directory mentioned earlier is an excellent example of a whole book full of hints. The number behind someone's name has all the properties of a hint: It may be wrong, but when it is used, one finds out that it's wrong. The correct number can then be found through a forwarding address or directory information.

In distributed systems, there are many examples of hints. The current network addresses of all sorts of services are often kept as hints (which become obsolete when a server crashes or migrates). The absence of a carrier on a broadcast network cable is a hint that the cable is free and that no other station is currently sending. If this hint proves wrong, there will be a collision.

Hints are often very easy to use and can result in staggering performance improvements in distributed systems. This makes hints a vital technique in designing and building high-performance distributed systems.

Use stashing to allow autonomous operation

The word *stash* was first used by Birrell and Schroeder at the 1988 ACM SIGOPS workshop in Cambridge. I quote from Schroeder's position paper:

> 'Stashing' is just a name for a class of techniques many of us have used in our systems over the years. It simply refers to keeping local copies of key information for use when remote information is not available. By recognizing the general technique, however, we can discover systematic ways to apply it.
>
> The application of stashing I know so far is naming and authentication information. To stash name server information, a client always tries the remote service first. When the service is available (the usual case), then the answer is remembered locally in a stash. Whenever the service fails to respond, the local stash is used instead, even though it may be out of date. So a stash is like a cache, except it has strange replacement policy. For small, slowly changing information, like the rootiest information from a global naming hierarchy, stashing would be easy to manage.

When stashing is used, a client machine can continue to make some useful progress when it is separated from normally essential network services (such as authentication service, file service, name service). The client machine cannot always proceed — files that are not stashed are not accessible when the file service is down — but at least some work can continue while the machine is cut off from networked services.

Exploit locality with caches

The cache is yet another way of keeping local copies of remote data. Stashed information may be stale and hints may be wrong, but cached information has to be correct. A cache is a local copy of remote data. Sometimes the remote data is updated more slowly than the cached data; sometimes both remote data and cache are updated simultaneously. This depends on the nature of the data, the way the data is shared and the frequency of updates.

Caching, stashing and using hints are three techniques for improving system performance by avoidance of remote accesses. Without them, distributed systems are almost necessarily painfully slow: a distributed system does more than a centralized one and the data and services are farther away. With caching, remote data can be accessed with virtually the same efficiency as local data.

When information is used from the cache, one has to make sure that the cached information is up to date. Cache-coherence protocols are used for this. Depending on the expected amount of sharing and the expected kinds of sharing, different

cache-coherence protocols may be appropriate. In Part IV about file systems and Part VI about replication, caching and cache coherence will be discussed further.

Use timeouts for revocation

There are many cases in distributed systems where a process locks up some resource while using it. When the process or its host crashes, the resource may be rendered inaccessible to others. As a general principle, it is a good idea to use timeouts on the locking up of resources. Clients must then refresh their locks periodically or they will lose them. The refresh frequency must be short enough and the timeout period long enough that the likelihood of having the lock taken away before it can be renewed is negligible.

One example where such a mechanism is useful is in servers using caching. Timers can be used in several ways in this context. One way is to give a caching client a shared read lock or an exclusive write lock, depending on whether it makes read or write accesses (Burrows [1988]). If the client crashes, the locks automatically break, allowing other clients to access the file. If the server crashes, the client will notice when trying to refresh its lock. It can then stash the file until the server comes back to life. If another client attempts to set a conflicting lock, a refresh from the first client could be denied.

Another way of using timers in a caching server is to use timers on the expiration of validity of cached data (Lampson [1986]). Essentially, the service tells a caching client: 'This data is good until such-and-such a time.' This is not very useful in a file system, but works quite well in a name server that, for instance, maps human names to mailboxes.

Use a standard remote invocation mechanism

There are many ways of invoking remote services. Current Unix systems illustrate this abundantly. One can do remote login, run a single command on a remote machine, invoke a file transfer to or from a remote machine, or invoke a specialized service by sending a message to a *daemon* on the remote machine (for example, *finger, rwho, ruptime*).

This not only leads to security flaws (see the 'worm' incident described in Chapter 20), but also to ineffective interfaces and complicated systems. Distributed systems usually provide only a few remote invocation mechanisms, and some provide only one.

Mike Schroeder and I both work on a system with a single remote invocation mechanism, remote procedure call, and believe that it is sufficient. RPC is simple

and can be made to run very efficiently (van Renesse, van Staveren and Tanenbaum [1988]).

Only trust programs on physically secure machines

Machines that are not physically secure can be tampered with. They can be made to bootstrap a version of the operating system that has been tampered with (if the boot ROM should incorporate an authentication mechanism, the boot ROM can be replaced; then the system can be brought up from a boot server running on the machine of a malicious user). Unfortunately, no secure bootstrap protocols have been designed for diskless workstations yet, so, for the time being, any diskless machine that is not on a physically secure network is in danger of being bootstrapped by a malicious user's kernel, even if the machine itself is in a physically secure place.

When a corrupted operating system runs on a machine, no process running on that machine is safe. Processes can be traced and keys used for authentication or data encryption can be found. This is why trusted servers must always run on trusted machines, and trusted machines must always be kept in physically secure places.

Use encryption for authentication and data security

Anything that appears on the network stops being a secret unless it is encrypted. Most local network interfaces allow machines to set their own network address and to receive every packet on the network. Thus, network addresses cannot be used to authenticate services (even if they run on a secure machine), and passwords for authentication can only be used once, because they stop being secret the moment they are used. Authentication protocols have to be used which use encryption to protect shared secrets from becoming public knowledge.

Try to prove distributed algorithms

Formal correctness proofs are gaining popularity. A variety of techniques now exists, some of which can deal with quite complicated algorithms. Reasoning about the correctness of a sequential algorithm is hard, but reasoning about the correctness of a parallel algorithm is virtually impossible. Examples of formal methods can be found in this book. In Chapter 5, Needham describes the use of formal methods in proving the correctness of authentication algorithms. In Chapter 11, a theory of nested transactions is presented by Weihl. In Chapter 15, Weihl then discusses specification methods for distributed programs.

But correctness proofs by themselves are not sufficient to guarantee that distributed programs are correct. After all, algorithms deal with an idealized world; the program has to deal with the real world. The boundary conditions used in a proof may not correspond to the real thing. Unexpected failures may occur, which the model did not take into account.

Building a reliable distributed system requires a combination of techniques; using formal methods is one of them, but testing is another that must always be used.

Capabilities are useful

The almost universal standard for academic computing these days is Unix. It uses access control lists (ACLs) for protection. When one is used to this model for protection, the alternative model, capability lists, does not come readily to mind. Yet, there is much to be said for using capability lists in distributed systems, especially if capabilities are objects managed in user space.

One benefit of using capabilities is that by holding a capability a client proves certain access rights to an object without any need for further authentication. Another, even greater benefit is that, in a capability system, a protection mechanism comes readily made with each service that is implemented. Server code needs only check a capability presented to it for validity — which can be done by a standard library package — before carrying out an operation. No separate authentication procedures need be carried out, nor need a service keep records of who can access what objects and how.

5 References

A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder [April 1982], *Grapevine: An Exercise in Distributed Computing*, Communications of the ACM **25**, 260–274.

A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder [February 1984], *Experience with Grapevine: The Growth of a Distributed System*, ACM Transactions on Computer Systems **2**, 3–23.

M. Burrows [September 1988], *Efficient Data Sharing*, Cambridge University Computer Laboratory, Ph.D. Thesis, Also available as Technical Report No. 153, December 1988.

5 References

B. Lampson [August 1986], *Designing a Global Name Service*, Proceedings 5th ACM Symposium on Principles of Distributed Computing, Calgary, Canada, 1985 Invited Talk.

R. van Renesse, H. van Staveren and A. S. Tanenbaum [October 1988], *Performance of the World's Fastest Distributed Operating System*, Operating System Review **22**, 25–34.

J. H. Saltzer and K. T. Pogran [1980], *A Star-Shaped Ring Network with High Maintainability*, Computer Networks **4**, 239–244.

J. Sventek, W. Greiman, M. O'Dell and A. Jansen [1983], *Token Ring Local Networks — A Comparison of Experimental and Theoretical Performance*, Lawrence Berkeley Laboratory Report 16254.

A. S. Tanenbaum and R. van Renesse [December 1985], *Distributed Operating Systems*, ACM Computing Surveys **17**, 419–470.