Distributed Operating Systems
State-of-the-Art and Future Directions

Sape J. Mullender

CWI, the Centre for Mathematics and Computer Science, Amsterdam,
and Computer Laboratory, Cambridge University

1. INTRODUCTION

Computing has become only a very minor task for modern computers. These days, the computer is mostly used for information storage, transformation and communication. Text processing is an excellent example of modern computer usage.

The need for better man-machine interaction has dictated much higher communication bandwidths between machine and user. Computer prices rapidly declined, while communication bandwidth remained expensive. This development naturally led to decentralized computing using personal computers and work stations with bit-mapped displays and pointing devices.

Disk storage technology still suggests large central disks because of their better price/capacity ratio and their performance characteristics: Accessing a remote fast disk over a local network can be made faster than accessing a local slow disk. Similar arguments apply to keeping other resources in a central place. Examples are high-speed laser printers, phototypesetters, tape drives, and number crunchers.

These developments have led to fairly typical configurations for general-purpose distributed systems: a fast local-area network, connecting work stations, file servers and printers. Some systems have a *processor pool*, a collection of fast processors which can be temporarily allocated for compute-intensive jobs, such as compilations. Many systems have gateways giving access to services over wide-area networks.

Several operating systems for such configurations are already commercially available. In these systems, however, the user (or the system manager) must still exercise explicit control over placement of files, over local or remote execution, etc. The state-of-the-art for commercially available systems is centralized systems with a few add-ons for remote operations. Systems like this are usually called *network operating systems*.

A distributed operating system is an operating system that runs on a large number of interconnected machines, but presents to the user an image of a single very powerful machine. Today, distributed operating systems only exist in research environments, although there are one or two exceptions (The Apollo Domain system is commercially available; systems such as V and Chorus are distributed in academic circles).

Distributed systems, having replicated hardware, provide a potential for fault tolerance and the exploitation of parallelism. To realise fault tolerance, files are replicated on several servers so that crashes do not affect the availability of files; processing is structured so that after a crash the interrupted work can be redone without causing inconsistencies. Parallelism is achieved by splitting up jobs in a number of parallel processes that communicate and synchronize by exchanging messages.

In this paper, we shall describe the state of distributed systems research and we shall attempt to identify future trends for research in the area. Sections 2, 3 and 4, will address communication, fault tolerance and parallism, respectively. Section 5 will describe typical distributed systems structures. Section 6 gives an overview of important distributed systems research projects. Future directions are discussed in section 7.

2. COMMUNICATION

Traditionally, communication between computer systems was modelled on the same lines as file i/o. Just like files are opened, read, written, or both, and then closed, communication *channels* were established (opened), read at one end, written at the other, or the other way round, or both read and written at each end, and finally the channel would be broken down (closed) again.

The network itself usually carries *packets* of data over the wire between stations. In wide-area networks, these packets pass through a number of intermediate stations on their way to their destination. Such networks are known as store-and-forward networks; each packet carries identifying information to tell intermediate stations where to send the packet next.

The channel that processes establish is thus an illusion created by the operating system, it is known as a *virtual circuit*. Virtual circuits are excellently suited for file transfer, for remote login sessions (connecting terminals to hosts), and for passing output from one process to another process as input.

For many years, this was the primary use of the computer network: remote login, file transfer, and electronic mail and news exchange. Most long-haul networks, which were designed in the seventies and early eighties, therefore adhere to the virtual circuit model of communication. It is a model that is simple to understand and it is convenient for the sorts of things that networks were used for.

With the coming of distributed computing this model has changed. The typical paradigm for distributed computing today is the use of *messages*. A message is a block of data that is sent from one process to another. Depending on the system messages have a fixed size, or a size in some range. Messages need not fit in a single network packet.

A fundamental difference between using messages and virtual circuit – as far as the operating system is concerned – is the way in which buffer management is handled. When virtual circuits are used, a relative complicated buffer management mechanism is needed for packet assembly and disassembly. This is so because reads to and writes from the virtual circuit may straddle packet boundaries. When messages are used, buffer management is much simpler, since packets contain data from a single message.

To illustrate the necessity of simple protocols and simple buffer management, consider an Ethernet connecting fast work stations, exchanging messages. When sending a small message (a hunderd bytes, say), the time spent in the Ethernet hardware is typically 200 μs, and time to schedule the receiving process (assuming light-weight processes) can be on the order of 250 μs. If the time spent in the message-passing protocols and the buffer management routines is much more than another 250 μs, then the efficiency of the message-passing mechanism is determined primarily by the efficiency of the protocol. It is important to realise that the number of machine instructions that can be executed in 250 μs is small, so protocols must be very simple to be fast in local-area networks. Note also that as networks become faster, the protocol overhead becomes even more the determining factor for the efficiency of the transport mechanism.

As another illustration, consider file transfer over a 100 Mbit/s network (e.g., FDDI). Let us assume that packets on such a network can be 10 Kbytes in size. The wire time for a packet is then 800 μs. The hardware may need another 800 μs or so to start transmission and to receive a packet, so the total network hardware time will be on the order of 1.5 ms. Again, if protocol times exceed this time by much, only a fraction of the available network bandwidth will be available for any single file transfer.

The fastest protocols using standard Ethernet hardware can do more than 1000 small messages per second between light-weight user processes and transmit around 5 Mbits/s between user processes using large messages. The fastest TCP/IP implementations, in comparison, do not achieve more than 3 Mbits/s.

Message passing is often embedded in protocols for *remote operations*: A process sends a request to another, the request is carried out and a response is returned to the originating process. Protocols, implementing this model are known as *request/response* protocols, *client/server* protocol, *message-transaction* protocols, or sometimes as *remote-procedure call* (RPC), although we shall see that the latter term is not quite appropriate. The process making the request is usually called a *client* and the process carrying out the request a *server*.

As it turns out, most of the communication needs in distributed systems are of a request/response nature: requests range from ‘are you still alive,’ ‘list files in a directory,’ ‘run this process,’ to ‘read this file.’ Even terminal i/o can often be conveniently modelled using request/response, especially when each subsequent interactive command (command interpreter, editor, text-processing system, mailer) can run on a different machine. When this is the case, it is much more convenient if the applications *fetch* their input from the terminal and send their output to the terminal. The terminal is treated as a *server* fielding and responding to requests for terminal i/o.

Request/response protocols are also used extensively for the implementation of remote procedure call mechanisms. In this mechanism, a process can call a procedure in another process on a different machine. The arguments are *marshalled* into a message which is then sent to the remote server. At the server, the arguments are unmarshalled, pushed onto the stack and the procedure is called. The results are returned in a similar way. Remote procedure call is thus a request/response protocol embedded in a programming language.

3. FAULT TOLERANCE

Replicated hardware offers an opportunity for distributed systems to provide some degree of fault tolerance. Building fault-tolerant systems, however, is quite difficult and very much a subject of ongoing research.

Fault tolerant file systems are especially important in reliable distributed systems. Most of the permanent or semi-permanent state of the system is stored on files. After a power failure, the file system often represents the complete recoverable state of the system.

Typical failures that file systems must survive are power failures, disk failures and processor crashes. During a power failure, the system naturally doesn't work, but after power is restored a quick recovery of the file system is important. During other failures, the system should continue to be useful. This means that important files – if not all files–must remain accessible, even if a disk or a file server crashes unexpectedly. To achieve this, obviously, files must be replicated so that, when one copy becomes inaccessible because of a crash, the other copy can still be used.

When a crash does occur, work is usually left half done and any crash recovery mechanism will have to deal with the problem of finding out exactly at what point the work was interrupted. In the general case, only the application can do this, but a distributed system can provide some assistance by reducing the number of states that the system can be left in after a crash.

Such assistance is often realised in the form of *atomic transactions*. Such transactions consist of a group of actions that are arranged in such a way that either all actions succeed, or – in the case of an error – all actions fail, leaving the system in a state as if none of the actions were ever attempted.

An atomic transaction is initiated by a *start transaction* operation, which returns a transaction identifier; then a number of actions are carried out, each one labelled with the transaction identifier, and, finally, the atomic transaction is either *committed* or *aborted*. If the commit succeeds, all actions are made permanent; if the commit fails, or if the transaction is aborted, all actions are undone.

There are several possibilities to implement atomic transactions. One way is to use *intentions lists*, lists that describe the changes to be made to the system when the transaction is committed. An intentions list has a boolean flag that indicates whether the transaction it belongs to has been committed or not. This flag is only set upon commit, when the list is complete. After commit, the intentions list is carried out. It is constructed in such a way that, if a crash occurs while the list is being carried out, it can be carried out again completely when the system comes back up. After a crash, uncommitted intentions lists are discarded.

Some systems implement *nested transactions*, atomic transactions within atomic transactions. These are very useful to obtain more fine grain control over failures. During the execution of a very large transaction, for instance, not all failures have to cause the abortion of the transaction as a whole, but can be recovered from within the transaction. Using subtransactions for this is very helpful.

Atomic transactions are now being implemented as basic distributed systems services. The Camelot project at Carnegie-Mellon University [Spector, Pausch, and Bruell, 1988] and the Quicksilver project at IBM Almaden [Cabrera and Wyllie, 1987] are examples of efficient transaction mechanisms in distributed systems.

Replication leaves objects accessible during arbitrary single-point failures, but creates complicated new problems: the consistency of replicated copies and the efficiency of accessing a replicated object. These two problems are closely related. It is relatively simple to keep copies consistent at a large cost in terms of efficiency, but it is very difficult to make access efficient while maintaining consistency.

Stable storage [Lampson and Sturgis, 1979] is a technique for maintaining replicated copies of disks in such a way that reads and writes of disk blocks are atomic; that is, a write succeeds successfully, or a write fails totally, preserving the previous information on the disk.

Another technique that makes replication easier is using *immutable objects*. Immutable objects can be replicated without the difficulties of multiple-copy update: immutable objects just never need updating. When immutable objects are used there must naturally be some other way to reflect change in the system. This is usually provided in the naming mechanisms: the name of an object is mapped by the naming mechanism onto the identity of an immutable object. Change is brought about by changing that map so that the same name refers to another immutable object. Thus, the problems of fault tolerance and atomicity are concentrated in one mapping service.

4. PARALLELISM

Distributed systems usually offer mechanisms for applications to exploit parallelism and use multiple processors to make programs run faster. Writing parallel programs is a fine art that nobody understands very well. The exploitation of parallelism, therefore, is limited to only a few areas.

An obvious area where parallelism can be obtained easily is in pipelining. When a process runs in several stages, each stage providing the input for the next, each stage can simply be made to run on a different processor. Buffered input and output over the network do the rest.

Parallelism is also easily obtained when a job consists of doing a number of independent things. For instance, when a program has to be compiled that consists of hundreds of files, it is usually possible to compile each of the files separately and in parallel. Compilation then takes as long as it takes to compile the biggest file.

This kind of parallelism can be detected by programs such as make in Unix. This program builds an application using a file that describes the constituing components and their interdependencies. Versions of parallel make [Baalbergen, 1986] often build a graph describing the partial ordering for things to be done and thus detects the parallelism.

Several language implementations attempt to provide mechanisms for expressing or exploiting parallelism. Basically, parallel languages come in two kinds: languages in which parallelism can be expressed, and languages that use parallelism internally so they run faster. Examples exist for both.

A number of fairly traditional imperative provide some simple support to create parallel subprocesses. Ada, Modula-2+, and even Algol68 are examples of such languages. A few languages have been especially designed for parallel applications. An example of such a language is Occam. Occam programs are usually executed on a network of Transputers, fast single-chip processors designed for easy interconnection.

Practically every distributed system provides support for parallel execution on different processors. Many systems even provide a 'parallel processing resource' in the form of a large number of processors that can be allocated to jobs as needed. With such a processor bank or processor pool, work stations can be used exclusively for highly interactive applications. Some distributed systems designers even clainm that work stations should be used only to run window-management software and possibly an editor; all other applications should run in the processor pool.

5. DISTRIBUTED SYSTEMS STRUCTURE

Many experimental distributed systems are being devveloped on top of an existing operating system. Usually, this is Unix. Although it seems a convenient approach to building distributed systems, the result of building on top of an existing syystem are usually disappointing: the resulting system is nearly always slow and it is hard to estimate what portion of the slowness is caused by the host operating system and what portion is inherent to the distributed system.

The performance of distributed systems built directly on the hardware is usually an order of magnitude better. These systems are nearly always characterised by small operating system kernels providing very limited functionality and additional functionality provided by application-level services. Distributed operating systems typically implement the mechanisms for process managemment and interprocess communication.

In most of the important distributed operating systems kernels [Cheriton *et al.*, 1986; Mullender and Tanenbaum, 1986; Rozier *et al.*, 1988], the interprocess communication is usually blocking and parallelism is achived through leightweigth threads of ontrol sharing an address space. Semaphores or something similar usually provide synchronization between these threads. This particular system structure is one of the few that has survived many experiments with all sorts of constructs for communication and parallelism in many different projects. It is one of the few structuring mechanisms that allow programmers to deal with parallel applications and also allow an efficient implementation.

File systems differ very much between systems. One reason is that, after all these years of file systems development, designing a good file system is still something of a black art. Very few really high performance file systems exist and even fewer exist that are distributed. Another reason is that the technology in this area still changes. Memory has become much cheaper making extensive caching possible. Processors have become much faster, making the performance gap between processor and disk larger. This makes the disk more of a performance bottleneck.

A third reason for the differences between file systems is the projected usage of the file system. The Andrew file system [Howard *et al.*, 1988], for instance, is a file system designed for the use in an academic environment with many hundreds of users. The file system does whole file transfer on open and close to reduce the load on the servers. This works, because in the Andrew environment, nearly all files are small. The Amoeba file server [Mullender and Tanenbaum, 1985], in contrast, has highly structured files to enable an efficient implementation of optimistic concurrency control.

The trend in distributed file systems seems to be towards caching file systems; file systems designed to make efficient use of large memory caches and still maintain a proper degree of consistency. To realise this, quite a few file systems designers have decided to use immutable version of files, combined with a version replacement mechanism.

6. FUTURE DIRECTIONS

Predicting the future is easy. Doing so with any accuracy is very hard. This section is the author’s attempt to predict the future. It is unlikely that all predictions come true, but some of them probably will; which ones those are, only the future can tell.

Wide-area networks will employ a global addressing scheme and a small set of standard protocols. Communication setup between any two sites in the world will be very much like the setup of a telephone call. Local-area networks will remain largely private; they will be connected to the wide-area network through gateways. The gateway will provide firewalls preventing local 'accidents' to penetrate into the wide-area network, and they can prevent intruders from penetrating into the local network from the outside. Gateways will also act as protocol converters so that efficient highspeed protocols can be used locally, tailored to the applications for which they are used, and standardised efficient long-haul protocols can be used in the wide-area network.

Distributed systems will mostly be object oriented [Lazowska *et al.*, 1981] and based on the service model [Mullender and Tanenbaum, 1986], that is, objects will be managed by services, and clients will do operations on those objects through the mediation of server processes. The system will support a name space for services and objects. Application programs will never have to use network addresses directly.

Workstations will have several powerful processors, a high-quality display, a keyboard and various other devices for interaction: video camera, microphone, loudspeaker, pointing device, drawing device. The work station will be mainly an interaction device; processing and information storage will take place elsewhere.

There will be a place for an air-conditioned computed room. File storage will be located there and also processor farms. It is on the processors in these farms that most of the computations will take place. The typical computer in a processor farm will be a multiprocessor, with a large (hundreds of megabytes) shared memory, accessed through coherent caches. Multiprocessors will be interconnected by one or several high-speed networks.

Air-condtioned computer rooms are needed for several reasons. Very fast processors dissipate a lot of power and generate a lot of heat. They will be noisy because of the fans that are used for forced cooling. Nobody want one of those in the office. Another reason is that, for fast distributed systems, communication delays must be short; the machines must therefor be physically close where that is possible.

Multimedia applications will become important in distributed systems. Techniques for video-commpression, high-speed local- and wide-area networks, and better distributed operating system kernels have now made real-time handling of video, voice and data in a distributed system feasible. Much research is still required in the areas of user interfaces, networking, protocols, real-time high-speed file systems, and the organization of the operating system to deal with multimedia.

7. REFERENCES

E. H. Baalbergen [1986].
Parallel and Distributed Compilations in Loosely-Coupled Systems.
Proceedings of the Workshop on Large Grain Parallelism, Providence, RI, October 1986.

L. F. Cabrera,and J. Wyllie [1987].
QuickSilver Distributed File Services: An Architecture for Horizontal Growth.
RJ5578, Computer Science Department, IBM Almaden Research Center, 1987.

D. R. Cheriton, E. D. Lazowska, J. Zahorjan, and W. Zwaenepoel [1986].
File Access Performance of Diskless Workstations.
ACM Transactions on Computer Systems 4 (3), Aug 1986.

J. H. Howard, M. J. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West [1988].
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6 (1), 1988.

B. W. Lampsonand H. Sturgis [1979].
Crash Recovery in a Distributed Storage System.
Xerox PARC, Palo Alto, CA, 1979.

E. D. Lazowska, H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal [1981].
The Architecture of the Eden System.
Proceedings Eighth Symposium on Operating System Principles : 148–159, December 1981.

S. J. Mullenderand A. S. Tanenbaum [1985].
A Distributed File Service Based on Optimistic Concurrency Control.
Proceedings of the 10th Symposium on Operating Systems Principles : 51–62, Orcas Island, WA, December 1985.

S. J. Mullender and A. S. Tanenbaum [1986].
The Design of a Capability-Based Distributed Operating System.
The Computer Journal 29 (4) : 289–300, 1986.

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser [1988].
CHORUS Distributed Operating Systems.
Report CS/Technical Report-88-7.6, Chorus Systèmes, Paris, Nov. 1988.

A. Spector, R. Pausch, and R. Bruell [1988].
Camelot − A Flexible, Distributed Transaction Processing System.
Proceedings Compcon 88 : 432–437, San Francisco, CA, February 1988.