Distributed Operating Systems

Sape J. MULLENDER *

Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

In the past five years, distributed operating systems research has gone through a consolidation phase. On a large number of design issues there is now considerable consensus between different research groups.

In this paper, an overview cf recent research in distributed systems is given. In turn, the paper discusses overall system structure, protection issues, file system designs, problems and solutions for fault tolerance and a mechanism that is rapidly becoming very important for efficient distributed systems design: hints.

An attempt was made to provide sufficient references to interesting research projects for the reader to find material for more detailed study.

Keywords: distributed operating systems, distributed file systems, fault tolerance.

Sape J. Mullender was born in Amsterdam. He has been at the Vrije Universiteit in Amsterdam from 1970 to 1983, first as a student of mathematics and computer science, later as a staff member, teaching programming and researching distributed operating systems.

His PhD. work under the supervision of Prof. Andrew S. Tanenbaum consisted of the design of the Amoeba Distributed Operating System, which is now being implemented.

Since 1984, Sape Mullender works at the Centre for Mathematics and Computer Science (CWI) in Amsterdam as project leader of the Distributed Systems Group. He is now at the DEC Systems Research Center in Palo Alto as a visiting scientist for a six month's period.

* Current address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.

1. Introduction

Advances in micro-electronic technology have caused a minor revolution in system architecture. Processors and are now so cheap and small that it has become practical to build computer systems out of many processing elements. Sometimes this is done by having several processors share a common theory, and sometimes it is done by giving each processor a private memory and providing communication facilities through a network.

If there are two different programs to be run, two processors are evidently more powerful than one: The work can be divided. But this is not so evident if there is only one program to be run. It is then much harder to put the available parallelism to use. Traditional system design methods and software engineering principles do not provide adequate methods of splitting up algorithms in independent parts which can be executed in parallel. Building distributed systems is easy. Using them is hard.

Potentially, systems built up of many processors are more reliable than traditional computers with a single CPU. If the processor fails, the system comes to a halt. In a distributed system, this is no longer necessary. Every single component of the system could be replicated, so that, no matter what component fails, a subsystem is left behind that can be made to work. If one processing element fails, others can take over the work. If a disk fails, a copy of the information could still be available on another disk.

As it turns out, designing software that exploits this fault-tolerant property of such a configuration is surprisingly difficult. Standard techniques for software development are all based on the assumption that the underlying hardware is infallible. This is a perfectly proper assumption in traditional systems, where, if part of the system fails, the whole system stops working, but it is no longer true in a distributed system.

Distributed systems research concentrates on the problem of structuring the hardware and designing the operating system software in such a way that we can profit by the architecture's two most important potentials, parallelism and fault tolerance.

Distributed systems research shows that there are many ways in which the problems can be attacked. Some systems are structured through their communication primitives, others through their language constructs, and others again through the underlying operating system. In this paper, we shall describe some recent work in distributed operating systems, concentrating on significant and unusual approaches. We shall look at system structure, protection, mechanisms for fault tolerance, and file systems. The last section will describe some of the significant trends in distributed systems work.

2. System Structure

Distributed systems have evolved from batch systems and time-shared systems, and, therefore, early distributed systems still had the *data-stream* model firmly in their communication systems. Interprocess communication was designed to resemble reading or writing magnetic tapes: First a virtual circuit was opened, then one wrote (or read) data for some time, and when the end was reached, the circuit was closed again.

This model was quite efficient, actually, when networks were still slow compared with processors: The explicit set-up of a virtual circuit allows the communicating parties to negotiate the allocation of buffer space to guarantee a certain throughput, and to streamline communication by allowing several packets of data to be en route at any time.

In modern fast local-area networks, with communication bandwidths of 10 Megabits per second, or more, such techniques no longer work. They actually tend to slow down communication. The extra processor cycles needed for optimizing network bandwidth by streamlining communication consume more time than is gained.

Protocols for use in distributed systems must be optimized for speed more than for anything else. Where conventional systems read files form the local file systems, distributed systems will often have to fetch files from a separate file server. Where in conventional systems all processing is local, in distributed systems processes may run on different processors for additional speed; their communication should not slow things down again. The user is seldom prepared to pay for additional flexibility by degraded performance.

Many successful distributed systems use the *message transaction* model: One process, the *client*, sends a *request* to another process, the *server* [1]. The server carries out the request and returns a *reply*. Different systems use different message lengths; some have fixed-size messages, others have variable-size messages, but in most of these models there is this request/reply pair in one form or another.

The message-transaction model is simple to implement and it can be made extremely fast.* But it is also easy to understand. There is a much better synchronisation between the actions of a program (e.g., *send request*) and the actions of the communication mechanisms (e.g., sending a message) than there is in a byte-stream-oriented protocol. In a byte-stream protocol, the actions of the program (e.g., send k bytes) are not synchronized with the actions of the communication primitives (e.g., send an n-byte packet). Explicit synchronization primitives have to be provided such as an *end-of-message* character, or a *flush* command.

Far more important than this difference between message transactions and byte streams is the difference between the two in *failure semantics* – the difference in what happens when a crash occurs in the system, or when an intermediate node goes down. In the connection-oriented model of the world, the communication primitives distinguish between a crash in the communication system and a crash in the end system. End-system crashes are no business of the communication mechanisms. Report of delivery of a message to a remote process is semantically meaningless, because the process may crash before it has had time to look at it.

The failure semantics of message transactions are much better: In the normal case (i.e. no crashes), the client receives a reply from the server. To the client this is a true end-to-end acknowledgement, because reception of the reply not only indicates successful reception of a request message, but also successful processing of the request.

* Our own system, the Amoeba Distributed System, obtains a user proces to user process continuous throughput of 500000 bytes/second, using full 32Kbyte messages and a minimum transaction response time of 10 ms. We do not know any distributed system using virtual circuits that achieves more than one third of this speed using off-the-shelf network interfaces as we do.

In case of a failure (i.e. no reply is received), the client cannot tell what went wrong: the request may have been lost, the server may have crashed, or the reply might have gone awry. No protocol exists that guarantees to find out which possibility occurred; there are always scenarios where the protocol doesn't know or reports wrong, so there is no point in attempting it.

Sometimes the point is made that acknowledgements that are not end-to-end increase the efficiency of the protocols. This may be true in unreliable and slow wide-area networks, but in fast local networks, communication errors are so rare that the overhead of the extra mechanisms may well *decrease* the average efficiency of the protocols.

Early distributed systems often had *asynchronous* communication primitives. Asynchronous primitives have separate calls for *initiating* communication and for waiting until that communication *completes*. There would, for instance, be a call *receive* which takes a message buffer as parameter, which would tell the kernel "If a message arrives, put it there; meanwhile, I'll continue doing some work," and a separate call, *wait* or *status*, which would be used to wait for the reception to complete or enquire whether it had completed already.

The philosophy behind these asynchronous calls was that they provide a mechanism for parallel processing, but experience has shown that it was not always practical.

Asynchronous calls cause more operating system overhead, because of the extra user/kernel interactions (at least one for initiating the call and one for waiting), but they also make things very complicated for the programmer. This is especially noticeable in the code for servers that serve several clients simultaneously. To exploit the parallelism offered by asynchronous calls, the code often ends up as a finite-state machine: at the top of loop the program waits for an event, finds out the task the event belongs to and executes a case statement on the state of that task and the event. Needless to say this hardly produces clear and modular code.

It is because of the inefficiency of asynchronous calls and the difficulty of using them that distributed system are now usually equipped with synchronous communication primitives. Naturally, it is no longer possible to obtain parallelism out of the communication mechanisms, so another mechanism is needed to obtain it.

This mechanism often consists of implementing parallel processes, in such a way that it is possible to have very many of them and that process switching and scheduling is cheap. In order to do parallel processing, one thus creates many of these *light-weight processes* which can then do blocking message transactions with remote processes.

Usually, groups of light-weight processes share one address space. Such a group is then called a *team* (V-System [2]), or a *cluster* (Amoeba [3]). This makes the implementation of services quite straightforward: All data structures – such as file tables and block caches for a file server – can be shared in the common address space, while each process can serve one request at a time. By creating enough light-weight server processes to start with, any amount of parallel processing can be obtained. The one remaining snag is that there can be race conditions when two light-weight processes concurrently access a common data structure. This problem can be overcome, however, by introducing *condition variables, monitors* or *semaphores*, or by scheduling processes only on blocking communication system calls.

3. Protection

Two approaches to protection are common in distributed systems research: In one, the system is treated as a closed system, with a trusted operating system kernel in each machine. All communication passes through the kernel, the kernel provides authenticating information and checks permissions, and the kernels trust each other. This model is based on the traditional operating system concept, it is simple, and is quite practical in many environments.

In the other model, a basic assumption is that workstations cannot trust the information from another workstation without making sure (usually by means of encryption) that the authenticating information has not been tampered with. The reason for this lack of trust if that it is often all too easy to re-boot a workstation with another, untrusted, version of the operating system to evade any protection mechanisms that reside there.

In the first model, it is, of course, necessary to assume that the operating system cannot be tampered with and that no process can avoid the protection mechanisms. Furthermore, the communication channel between kernels must be secure against passive or active intruders (wire tappers). This can usually be guaranteed by using encryption on the channel between two kernels. The whole model is well-understood and easy to realise, assuming a secure kernal can be built and workstations cannot be secretly re-booted.*

The second model is much more interesting from a research point of view, and probably more realistic as well. A process can only distinguish itself from another one by using some *secret* that other processes do not have; using this secret, messages can be generated or understood that other processes cannot generate or understand. An example of such a secret in traditional operating systems in the *log-in password*, or – as present in some systems – a *file password*. In distributed systems, typical secrets are *encryption keys* or *capabilities*.

Conceptually, the *effect* of a protection mechanism can be modelled by a matrix, M, indexed by *user* and *object* (or *resource*). The operations allowed by user i on object j are given in M_{ij}. This conceptual model is usually implemented in one of two ways. The first is to store with each object a list of users' rights to the object; that is, to keep the object's column of M with the object. This list is referred to as an *access control list* or ACL. The other is to give each user a list of all objects with the user's rights on them; that is, to give each user the corresponding row of M. This list is referred to as a *capability list*.

When ACLs are used, servers that manage objects and control the access to them must have some way of establishing the client's identity. Similarly, clients must have a way of making sure that requests to do operations on objects are fielded by a genuine server and not by some imposter. An *authentication mechanism* is needed to determine the identity of the client to the server and of the server to the client. When capabilities are used, clients still need to have a way of ascertaining the authencity of the server, but the server need not know the identity of the client – it is enough that the client can produce a capability for the object.

* This assumption makes it almost mandatory that the workstations must be in a computer room or terminal room, where they can be physically supervised, or that the bootstrap ROMs are modified to refuse to bootstrap from a user-supplied binary.

Instead, a mechanism is needed that prevents clients from forging capabilities. A mechanism for this can be easily be constructed: A server can build one by concatenating a server identifier, an object number, a bit map indicating the rights and a check field. The check field can be computed using some function which takes the three other fields plus a random number – stored along with the object – as input, scrambles the bits throughly, and produces the check field. The random number is kept as a secret within the server, so only the server can compute capabilities.

Authenticating clients and servers can be done with the help of an *authentication server*. It must be trusted by the clients and servers that use it, and provides encryption keys for the communication between a specific client/server pair. Public-key encryption would be even better, because it provides a mechanism where only one key pair is needed for every client or server. With n clients and servers, n^2 conventional keys are needed, while only $2n$ keys are needed when public-key encryption is used. Unfortunately, public keys are very large (a few hundred bits at least), encryption is slow, and therefore not practical.

A few systems rely on a different concept of protection. Two interesting examples are the ITC system at CMU [4] and the Amoeba system at CWI in Amsterdam [3]. The ITC distributed system consists of hundreds of workstations at which students do their programming assignments. In addition to those workstations, a number of dedicated servers give students access to a large shared file system, printing services, etc. The protection concept is that the central services can be kept under lock and key in the computer room so they can be trusted, while the student workstations are out in terminal rooms and in private offices, so they can be tampered with. The trusted environment (the servers) is called *Vice* and the untrusted environment (the workstations) is called *Virtue*. Nearly all communication takes place between *Vice* and *Virtue* and *Vice* servers also act as authentication servers.

The unconventional protection mechanism of the Amoeba system [5] is also based on the notion that the operating system of workstations in private offices or terminal rooms can easily be tampered with, but also on the notion that encryption of all data for protection might slow down the system too much (there was no room for encryption hardware in the budget). The assumption was made, however, that in student environments, tampering with the hardware is an order of magnitude less likely than meddling with operating system software. The Amoeba protection mechanism is based conceptually on a simple device, inserted between each workstation and the network, which filters incoming and outgoing messages. Communication is based on *ports*: A process sends a message to a port and another receives it on a port. Each port has two names, one for sending to it, and one for receiving on it. The little interface box matches port names and only lets messages through if the receive port has been given to it by the potential recipient. Names of ports are essentially large random bit patterns, large enough so they can't be forged. By keeping its receive port secret and publishing its send port, a server can go into business without fear of being impersonated by a malicious client.

4. File Systems

A reliable file system is an important aspect of fault tolerance and much research has been done in the area of building file systems with things like replicated storage, concurrency control mechanisms, local caches of remote data, version control, etc. [6].

There is no general consensus on what a file system should do exactly as this depends very much on the environment in which it is used. Interesting new types of file servers are the ones that do *whole-file transfer* [7,8]. Files are always read and written as a whole. This type of file server is very efficient in environments with reasonably small files and with low concurrency and is used in universities for storage of student files.

Another type of file server that is seen more and more is one where files have *versions* [9]. A version of a file, after it has been made, becomes immutable, and changes to the file are represented by a sequence of versions, each new one based on the previous one. This type of file server works well on optical disks, which can only be written once. As an optimization, often only the differences between a version and the previous one are actually stored.

Some file systems have an elaborate file *naming* mechanism, where files can be named in a hierarchical naming tree or graph. Other file systems only implement *unique identifiers* for naming files or versions [10]. Another service (*directory service*, say) can implement any naming mechanism on top. The latter method gives the flexibility to implement several naming structures on top of a single file system.

One general trend in file system research is observable: There is considerable interest in *caching*. File caches are necessary for making file systems efficient, but they are very much in the way of concurrency control mechanisms. The version-type file systems appears to do best in combining the two.

5. Fault Tolerance

For a large part, fault tolerance is what distributed systems research is about. Having two or more copies of everything makes it possible, at least in principle, to continue operations when failures occur. The biggest problem is to discover failures and to recover from them.

Failures come in categories. The worst failures are often referred to as *Byzantine failures*,* failures where parts of the system behave incorrectly and sometimes even maliciously so. The file server, for instance, could lie about the contents of a file. Algorithms that work correctly in environments with Byzantine failures are Byzantine algorithms; they are very robust but usually very inefficient as well.

Most distributed systems are based on the assumption that processes exhibit so-called *fail-stop* behaviour. When a process fails, it stops; that is, processes work normally, or not at all. Under this assumption, it is much easier to construct algorithms that continue to work when a limited number of processes fail.

In addition to fail-stop behaviour, it is useful to be able to group the actions of a process in such a way that whole groups of actions get done correctly until a failure occurs and that the last group of actions is either completely and correctly done,

* Names after the problem of the Byzantine generals [11]: n generals, of which k generals may cheat, must make a binary decision. Design a protocol that makes such a decision in a way that all generals know the correct outcome in spite of the efforts of the cheating generals.

done, or none of the actions in the last group is done. Such a group is an *atomic action*. Either all of an atomic action succeeds correctly, or none of it; there is no middle way, no half-done atomic actions.

A (virtual) disk, whose blocks can be read and written atomically, is referred to as *stable storage*. It is implemented with two identical disks, and the implementation is based on the principle that disks have a *weak atomic property* [12]. When a block is written, and the writing hardware or software fails, the block is in one of three states:
– the actual writing had not started yet and the disk is still unmodified;
– the actual writing had already finished and the write completed correctly;
– or it was broken off half way, but the disk is in a *detectably bad state*, the hardware CRC checksum is in error and this is detected when it is next read.

Whenever a block is written, it is first written on one disk, and only after the write has finished correctly it is written to the other disk. If a failure occurs, at least one of the disks is always in a correct state (a *before* or an *after* state), and the stable storage system can recover when it comes up again. If the crash occurs exactly between the two writes, both disks are readable but different. When the system comes up, one copy can be chosen and written over the other.

Using, for instance, stable storage to implement atomic I/O for disk blocks, atomic I/O for multiple blocks can be implemented. Using that, it is possible to build arbitrary atomic actions by pretending to carry out the actions and to store the results (modifications of *the system*) on an *intentions list*. When the atomic action ends, the modifications are first flagged permanent (by atomically changing one bit), and then introduced into the system by executing the intentions. The intentions list is structured so it can be executed partly any number of times and once correctly.

Atomic actions can be further structured by allowing *nested* atomic actions [13,14]. Concurrent atomic actions are hard to implement, especially if one wants as much concurrency as possible. Mechanisms to implement concurrent atomic usually use locking [15], and sometimes optimistic concurrency control [16,17].

6. Hints

A notion that has emerged in distributed systems research that is important enough to deserve a separate section is the *hint*. The idea of using hints came from Butler Lampson [18] who used it in the file server of the Alto operating system. Since then, it has become a very popular mechanism to speed up distributed operations that might otherwise be too slow for comfort.

In [19] Lampson describes a hint as the "saved result from a previous computation." Hints are very much like cache entries, but there is one essential difference: *Hints may be wrong*. This makes hints no less worth while as a mechanism, provided there is a *backup* mechanism that will prevent harm from using a faulty hint and correct the hint.

The idea is best illustrated by an example. A send port in the Amoeba system looks like a random bit pattern. It provides no clue about the whereabouts of the receiving process. To make it even more difficult, there can be several processes receiving on a single receiver port, and the system is allowed to deliver a message to any one of them. Furthermore, processes are mobile, processes can crash and new processes can be created. However, in practice, the location of most processes does not change very much. A sender process, therefore, keeps hints consisting of send ports and network addresses. When a process sends to a send port and there is a hint for that send port, the system sends the message to the indicated network address. But every once in a while, the receiving process has migrated, or crashed, and is no longer there. In Amoeba, the *backup* mechanism consists of a message sent by the network interface of the wrong recipient back to the sender, saying "unknown at this address." The system then finds the correct location using a broadcast mechanism (or a distributed name server in the wide-area network).

Hints can be used in many cases. Hints are used in file systems to find the next block of a file without having to consult an index on disk, saving a disk access (if the file changes, the hint becomes incorrect; the backup mechanism is often the header of the next block). They are used in the routing tables of store-and-forward networks. The absence of a token for some period is a hint that the token was lost in token-ring network (it is a more hint, because another station may just have generated a new one).

Hints have become important in distributed systems, where changes to the system state (whatever that is) are not immediately visible everywhere. It is extremely handy to be able to assume that something remote is in a particular state (it usually is) and that an operation will succeed if the assumption is right, but that some backup mechanism will come into operation if it is not.

7. Summary

Successful distributed systems are all simple systems. The interprocess communication mechanism is always simple and fast. Remote procedure call is used, or message transactions. Processes are usually light-weight, and communication calls are blocking. When a server crashes, the client process is usually responsible for recovery (usually by repeating the action on another server).

Protection is treated seriously enough as a research issue, but it is seldom put rigorously into practice. Only in student environments is some effort often necessary to make the system secure, but the Cambridge Distributed System, for instance, which has a perfectly secure *concept* of protection does not really use it; the system administrators rely on the good manners of the Cambridge student. Which works.

File systems are tailored for the environment in which the system is used. In environments populated primarily by students, file systems typically offer whole-file transfer for efficiency and simplicity. Few, if any concurrency control mechanisms are offered, because changing files are not often shared. In environments with much concurrency, one would expect very sophisticated file systems, but this seems to be only rarely the case. Actually, not many distributed systems are in operation in demanding environments in the first place. Usually, distributed file systems offer files as a linear sequence of bytes and locking for concurrency control.

Atomic actions are an important tool for implementing fault tolerance. Structuring fault-tolerant systems, or better still, structuring fault-tolerant systems in such a way that they are easy to use, will need more research. Interesting work is going on at Cornell, where the ISIS system is being developed [20]. It executes ordinary programs in a fault-tolerant way.

Albert Einstein is reported to have said "Everything should be as simple as possible, but no simpler." This certainly applies to system design, and, if possible, even more to distributed system design. All too often, systems are made too complicated and become inefficient and impractical because of their complexity. Almost everything designed by a committee has this property: X.25, Ada, TSO, to name a few. A distributed system is inherently faster than a centralized one: parallel processing can be exploited; but a distributed system is also inherently slower: files that used to be always on a local disk, are now often stored in the (remote) file server. Communication overhead creeps in everywhere. If the distributed system must be used, it must be at least as good as the old centralized system. The single most important way to achieving this is to make communication as fast as possible. And this can only be done by making the communication interface as simple as possible.

References

[1] A.Z. Spector: "Performing Remote Operations Efficiently on a Local Computer Network," Communications ACM, vol. 25, no. 4, pp. 246–260, April 1982.

[2] D.R. Cheriton and W. Zwaenepoel: "The Distributed V Kernel and its Performance for Diskless Workstations," Operating Systems Review, vol. 17, no. 5, pp. 129–140, October 1983.

[3] S.J. Mullender and A.S. Tanenbaum: "The Design of a Capability-Based Distributed Operating System," Computer Journal, vol. 29, no. 3, 1986.

[4] M. Satyanarayanan: "The ITC Project: A Large-Scale Experiment in Distributed Personal Computing," in Proc. of the Networks 84 Conference, Indian Institute of Technology, Madras, Oct 1984, North Holland, 1985. Also available as ITC Technical Report CMU-ITC-035.

[5] S.J. Mullender and A.S. Tanenbaum: "Protection and Resource Control in Distributed Operating Systems," Computer Networks, vol. 8, no. 5, 6, pp. 421–432, 1984.

[6] L. Svobodova: "File Servers for Network-Based Distributed Systems," ACM Computing Surveys, vol. 16, no. 4, pp. 353–398, December 1984.

[7] M. Satyanarayanan: "The ITC Distributed File System: Principles and Design," Proc. 10th Symposium on Operating Systems Principles.

[8] M.D. Schroeder, D.K Gifford and R.M. Needham: "A Caching File System for a Programmer's Workstation," Proc. 10th Symposium on Operating Systems Principles, December 1985.

[9] M. Fridrich and W. Older,: "The Felix File Server," Proc. 8th Symposium on Operating Systems Principles, Operating Systems Review, vol. 15, no. 5, pp. 37–44, December 1981.

[10] J. Dion: "The Cambridge File Server," Operating Systems Review, vol. 14, no. 4, pp. 26–35, Oct. 1980.

[11] L. Lamport, R. Shostak and M. Pease: "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382–401, July 1982.

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

[13] B. Walker, G.J. Popek, R. English, C. Kline and G. Thiel: "The LOCUS Distributed Operating System," Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review, vol. 17, no. 5, pp. 49–70, October 1983.

[14] E.T. Mueller, J.D. Moore and G.J. Popek: "A nested transaction mechanism for LOCUS," Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review, vol. 17, no. 5, pp. 71–90, 1983.

[15] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database Operating System," Communications ACM, vol. 19, no. 11, pp. 624–633, November 1976.

[16] H.T. Kung and J.T. Robinson: "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, vol. 6, no. 2, pp. 213–226, June 1981.

[17] S.J. Mullender and A.S. Tanenbaum: "A Distributed File Server Based on Optimistic Concurrency Control," Proc. 10th Symposium on Operating Systems Principles, pp. 51–62, December 1985.

[18] B.W. Lampson and R.F. Sproull: "An Open Operating System For A Single User Machine," Proc. 7th Symposium on Operating Systems Principles, pp. 98–105, 1979.

[19] B.W. Lampson: "Hints for Computer System Design," Proc. 9th Symposium on Operating Systems Principles, October 1983.

[20] K. Birman: "Replication and Fault Tolerance in the ISIS System," Proc. 10th ACM Symposium on Operating Systems Review, vol. 19, no. 5, pp. 79–86, December 1985.