SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(7), 657–675 (JULY 1991)

Design and Implementation of a Distributed File System

HSIAO-CHUNG CHENG AND JANG-PING SHEU
Department of Electrical Engineering, National Central University, Chung-Li 32054, Taiwan

SUMMARY

We introduce a new model for replication in distributed systems. The primary motivation for replication lies in fault tolerance. Although there are different kinds of replication approaches, our model combines the advantages of *modular redundancy* and *primary-stand-by* approaches to give more flexibility with respect to system configuration.

To implement such a model, we select the IBM PC-net with MS-DOS environment as our base. *Transparency* as well as *fault-tolerance file access* are the highlights of our system design. To fulfil these requirements, we incorporate the idea of *directory-oriented replication* and *extended prefix tab/es* in the system design. The implementation consists of a command shell, a DOS manager, and a recovery manager. Through this design, we can simulate a UNIX-like distributed file system whose function is compatible with MS-DOS.

KEY WORDS Crash recovery Distributed tile systems File replication Fault tolerance Network transparency

INTRODUCTION

With progress in the area of computer networks and operating systems, distributed processing has become more and more attractive in recent years. Many distributed systems have been proposed.[1-5] These systems differ in their goals, design philosophies and implementations, but there are still many other approaches worth trying. We shall concentrate on the design and implementation of a distributed file system. Furthermore, we wish to exploit the fault-tolerant potential of distributed systems. In a distributed file system the storage is distributed over the network. The failure of a few sites does not cause a disaster because there are always some sites still working well. Thus, if we replicate a file and distribute the copies over the network, the availability of the file is significantly enhanced.

Although fault tolerance is the nature of distributed systems, whether it can be used depends on the facilities provided by their file mechanisms. Such facilities must replicate the file and maintain the consistency of each copy automatically. Many researchers have been working on that, but their approaches are different. One of these is to support no mechanism for replication at all, as in the first version of Clouds,[6] V system[2] and Unix United.[7] Another scheme, introduced by Purdin, Schlichting and Andrews,[8] provides low-cost semi-automatic file replication and location *transparency* facilities for a network running Berkeley Unix. This can be

0038–0644/91/070657–19$09.50
© 1991 by John Wiley & Sons, Ltd.
Received 17 July 1989
Revised 10 October 1990

done through two mechanisms: *reproduction sets* and *metafiles.* Consistency between copies is achieved on a 'best effort' basis. Other systems, such as LOCUS,[1] Eden[9] and an extended version of UNIX United,[10] have mechanisms to handle file replication and consistency between copies despite failures. Sprite's mechanism for locating files (prefix tables) indicates some potential for replicating files, although the exact mechanism for maintaining consistency is not clearly described.

Our goal is to design a distributed file system with emphases on *fault-tolerant* features and network *transparency.* The base of our implementation is the IBM PC-net with an MS-DOS environment; we intend to simulate a UNIX-like file system on it. Our implementation will provide network users a more convenient tool to access network resources without knowing their locations. Furthermore, a fault-tolerant feature is included in our implementation. It will be like an MS-DOS extension with network capabilities and fault-tolerant file accesses. In fact, it is a distributed file system on the IBM PC-net environment.

AN ABSTRACT MODEL FOR FAULT TOLERANCE

Issues concerning fault-tolerance models

There are various kinds of errors that can occur in a distributed file system. To name a few: programming errors, operator errors, transaction errors, system errors and media errors.[11] Generally, they can be divided into two classes: software failures and hardware failures. Software failures are caused by incorrect programming or even ill-devised algorithms. Errors of this kind are very unpredictable and hard to deal with as they contradict the assumption that all the execution results will preserve the consistency of the file system. Therefore, we do not aim to correct such errors. On the other hand, hardware failures such as server crashes are more predictable within our model. Thus our fault-tolerant approach aims primarily at hardware failures.

The fault tolerance is enforced through replication. According to Yap, Jalote and Tripathi[12] and Bloch, Daniels and Spector[13] the distributed schemes (by contrast with non-distributed ones[14]) that use replication to support fault tolerance can be classified into three categories: the *primary-stand-by* approach, the *modular redundancy* approach, and the *weighted voting* approach. The primary–stand-by approach[1,15,16] selects one copy from the back-ups and designates it as the primary, whereas the others are stand-bys. Then all subsequent requests are sent to the primary copy only. The stand-by copies are not responsible for the service, and they only synchronize with the primary copy periodically. In case of a failure, one of the stand-by copies will be selected as new primary one, and the service goes on from the point synchronized most recently. Such a scheme does not waste resources on duplicated copies, although recovery from a crash is not instantaneous. A variation is *primary–secondary copies*[17] in which the modification requests are propagated to the secondary copies, but consistency between secondary copies is not guaranteed at any time and enquiries are not limited to the primary copy.

The modular redundancy approach,[5] also called *unanimous update,*[18] makes no distinction between the primary copy and stand-by ones. Requests are sent to all the back-ups simultaneously, and service is performed by all the copies. Therefore, it is fault tolerant provided that there exists at least one correct copy. In contrast to the primary–stand-by approach, the service continues instantaneously after the fault occurs, but it is costly to maintain the synchronization between the duplicated items, especially when there are many of them. Furthermore, when the number of replicas increases, the availability for file accesses decreases, because any update operation will lock all the replicas.

In the weighted voting approach,[19] all replicas of a file, called representatives, are assigned a certain number of votes. Read (write) operations are performed on a collection of representatives, called a read (write) quorum. Any read (write) quorum which has a majority of the total votes of all the representatives is allowed to perform the read (write) operation. Such a scheme enjoys the maximum flexibility in that the size of the read (write) quorum can change for various conditions. On the other hand, it may be too complicated and hence not be feasible in most practical implementations.

Our model

Our model is similar to the modular redundancy approach, but some modification is made to accommodate the advantages of the primary–stand-by approach. In our model, all copies of a file are divided into several partitions. Each partition functions as a modular redundancy unit. One of the partitions is selected as primary and the others are backups, just as in the primary–stand-by case. In this manner, we find a balance in the trade-off between the above two approaches. Intuitively, the choice is an eclectic one. Thus flexibility is enhanced while instantaneous forward progress is still retained to a certain degree.

In fact, this scheme can be seen as one that generalizes both the modular redundancy and primary–stand-by approaches. Let P denote the number of partitions, and W_i, $1 \leq i \leq P$, denote the number of replicas in each partition i. When P is equal to 1, this scheme reduces to the modular redundancy case. If all W_is are equal to 1, our scheme becomes the primary–stand-by case. Thus by adjusting P and each W_i we can reconfigure the system for different conditions at any time. The flexibility will thus be enhanced while the scheme still retains the instantaneous forward progress.

There are some problems involved in the model. First, how to choose a primary partition from all the equally capable partitions. We can choose one arbitrarily, or we can run an algorithm as complicated as *consensus protocols.* But the amount of information exchange in such an algorithm could be extremely large, which is not desirable. A simpler solution is to let the one that responds first be the primary partition. This also involves some consensus problems and communication overhead. Evaluations and comparisons between the above schemes are beyond the scope of our discussion. We just choose the simplest way: to predefine all the partitions as a linear order.

Secondly, there is the problem of how to enforce the single-copy semantics in each partition. Since the modular redundancy scheme is used in each partition, the single-copy semantics will be guaranteed by duplicating each update operation on the partition. However, in order to reduce the communication overhead of the client machines, all replicas in a partition will form a linear list together. The requests can then relay through the list, and the communication overhead is distributed over all the replicas. Thus a client is no longer responsible for handling all the call and returning messages, and the single-copy semantics is still enforced.

Thirdly, forward progress must be guaranteed. If there is any replica available when a fault occurs, the execution ought to continue as smoothly as possible. It is a simple job when the primary partition still has replicas available. But data consistency may be a problem if the available replicas are only in the back-up partitions. The most popular approach is to establish checkpoints [20] periodically. That is also the scheme we are using. When the failed server is resumed after a failure, the data in store would not be the most up-to-date. The data should then be corrected according to information obtained from the checkpoints.

Scenarios

In the following, we show how the model works. The group of replicas is divided into several partitions, as shown in Figure 1. The replicas (servers) in each partition in turn form a linear-order list. The ordering criteria can be the response time, the load of that server, or anything related to the system performance.

A service request is sent to the head of the replica list of the primary partition. If this request is an update operation, then the head replica must propagate the request to the next replica, which in turn sends the request to its next replica. All the update requests are queued. When a checkpoint is reached, the queued requests are relayed to the other partitions in order to maintain the consistency. Let us examine our model in the following conditions.

First, let us assume that failures are absent. Under normal conditions, only the first replica of the primary partition list will get the request. If no data are modified during the execution, the result will be sent to the client and only one request is made. If the request has updated some data, the request will propagate to the servers on the rest of the partition list. Each server will wait for the acknowledgement of its successor after the server has sent out its request. Upon receiving that signal from its successor, the server will send the acknowledgement to its predecessor. But from the viewpoint of the client, the request is made only once, and only one copy of the result is received by the client. The request and the acknowledgement sent by the client and servers are shown in Figure 2.

Secondly, there may be failures in the primary partition. This kind of failure can occur in two places: the first replica or the replica in the rest of the primary partition list. In both cases there are two types of failure-occurrence time: before or after the replica server propagates its request.

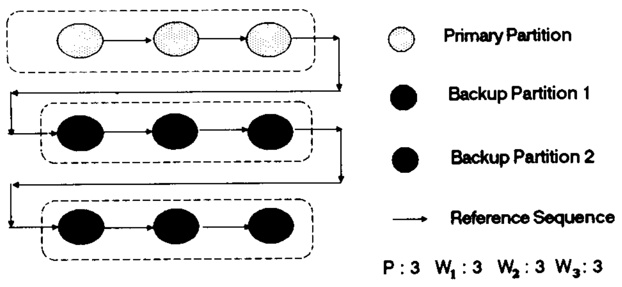


Figure 1. Partitions of replicas

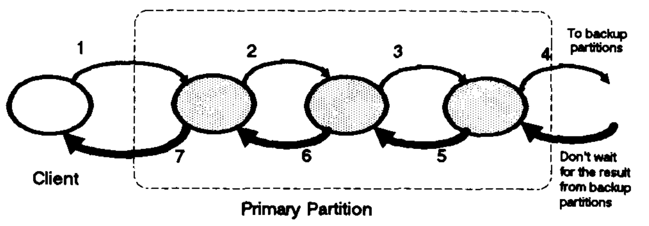


Figure 2. Messages sent in normal condition. The numbers indicate the message sequence of a request–service cycle. All servers in the primary partition will relay the request and wait for the result (acknowledge) of its successor. Servers in the back-up partitions do not synchronize with those in the primary partition

When failure occurs in the first replica before the request is propagated to its successors, the client will detect that the first replica server has failed, and the same request will be sent to the first replica server's successor. Then everything continues just as if no error has ever occurred, except that the client repeats the request and the result is sent directly from the next server to the client, as shown in Figure 3.

Now consider the case of failure occurring in the first replica after the request is propagated to its successors. If the first replica server fails after the request has propagated to the successor, the successor will receive the same request twice. Since the request carries with it a sequence number, the successor will execute the request only once, and the result is sent directly from the next server to the client, rather than to the first server, as shown in Figure 4. When failure occurs in the other replicas before (or after) the request is propagated to their successors, the working scheme is the same as when the failure occurs in the first replica except that the word 'client' is replaced by 'predecessor' above.

Finally, when all the replicas in the primary partition fail, as shown in Figure 5, one of the back-up partitions is selected to take over as the new primary partition. Nevertheless, how to choose it? We use a straightforward way. The partitions are given in a default order to reduce the complexity of the system implementation.

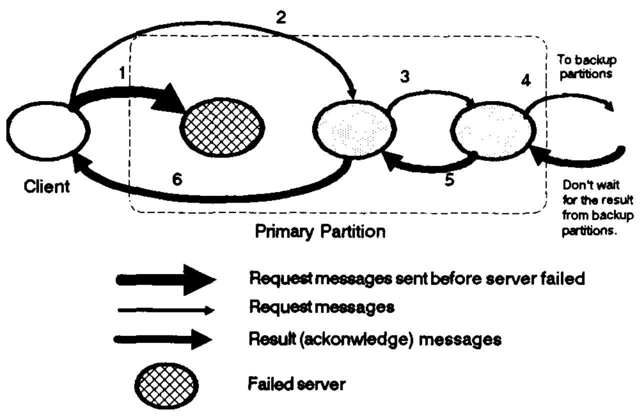

2
1
3
4
To backup partitions
Client
6
5
Don't wait for the result from backup partitions.
Primary Partition
Request messages sent before server failed
Request messages
Result (ackonwledge) messages
Failed server

Figure 3. Failure occurs in the first replica before the request is propagated to its successors

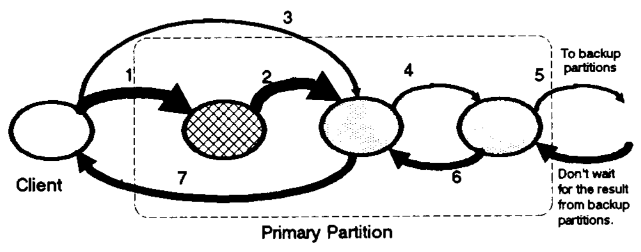


Figure 4. Failure occurs in the first replica after the request is propagated to its successors

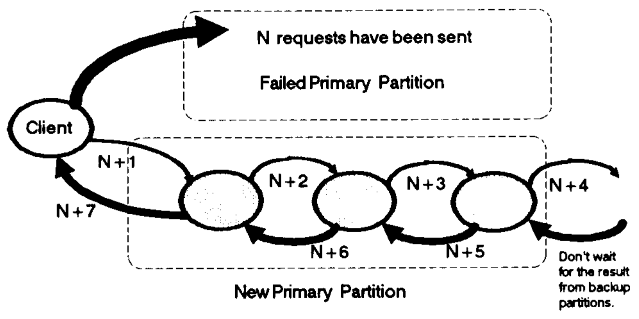

N requests have been sent
Failed Primary Partition
Client
N+1
N+7
N+2
N+3
N+4
N+6
N+5
Don't wait for the result from backup partitions.
New Primary Partition

Figure 5. Messages sent in the case where the whole primary partition fails

When a new primary partition takes over, the status of the file is restored to what it was at the latest checkpoint, and the execution then continues.

Recovery

Recovery from a failure is important for a fault-tolerant system. In contrast to fault detection and assessment, which are passive because they are not intended to change the system, [21] the recovery does produce changes in the system in order to accomplish the restoration; therefore it is active. Our recovery management has at least two responsibilities. One is to restore the file system to the latest consistent state after failures. The other is that the consistency among the replicas must be guaranteed after crashed servers return to service in the network.

The recovery strategies can be divided into two classes: the centralized approach and the distributed one. When traditional file systems are replaced by distributed file systems, the recovery mechanism is more complex because distributed storage imposes a big problem for consistency, and different kinds of errors are added by communication failures, server failures, and perhaps even failures of the recovery processes themselves. Since most of the recovery mechanisms come from database systems, our discussion about recovery relies on analogies with the ways in which database models handle recovery.

There is a way to classify recovery mechanisms according to the file-system states after the restoration. *Backward recovery* restores all the objects to the state before the atomic action begins, without regard to the current state. On the other hand, *forward recovery* makes use of a part of the current state to produce a new state that is error-free. We use backward recovery, as it is more straightforward. Atomic action [22] is a very important characterization of a distributed recovery model. An action whose updates can be performed without undoing the updates of other actions is called *recoverable.* In our model recoverable atomic action is assumed.

One problem arises when we handle storage recovery. Should we choose a new server and mark the crashed server obsolete after the server fails; or should we wait for the failed server to come back? The former approach can ensure that there are always available back-ups. The disadvantage is that the overhead of recreating a new back-up may be very high. Besides, making a whole server obsolete because of a single update failure seems to be unreasonable. As a result, we have chosen the second approach to reduce the overhead. Moreover, we think that the server storage is relatively stable, as it does not fail very often. In the condition that several back-ups exist, we prefer to wait for the server's recovery.

Our recovery mechanism is rather simple compared with other complex schemes. Refer to the example in Figure 1, where the responsibilities of the failed servers in the primary partition will be taken over by their successors. As shown in Figure 6(a), when the failed server is recovered, it will restore the modified data into a consistent state. If the whole primary partition fails, the data in the back-up partitions may be obsolete, because the most recent update operation may not have had time to reach the back-up partitions. However, the next partition of the primary partition is selected as the new primary partition in our scheme. When the failed servers are recovered, their states are updated to be consistent with those of the servers in the new primary partition, as shown in Figure 6(b).

DISTRIBUTED FILE SYSTEM DESIGN

The goal of our distributed file system is to provide users with transparent and fault-tolerant remote accesses to the file system. The approach we use to achieve fault tolerance has been described in the abstract model. As for *transparency,* in order to provide an interface just like the centralized time-sharing systems, four types of transparency must be satisfied: [23] *location transparency, replication transparency, concurrency transparency* and *failure transparency.*

To satisfy the constraints mentioned above and to implement the abstract model, we introduce several mechanisms which will be added to the IBM PC-net with MS-DOS environment. The result is a distributed file system that is functionally indistinguishable from ordinary centralized MS-DOS systems, at least at the shell-command level.

Directory-oriented replication

When fault tolerance is enforced through replication, the granularities of replication vary in size from file to disk. On one hand, a file (or even a record, though that is unrealistic if we consider the overhead) can be the unit of replication. Such an approach requires a mapping table for every replicated file. As the number of replicated files or their replicas increases, the overhead of maintaining such tables may cost too much. On the other hand, one disk or server can be dedicated to the back-up of another disk or server. Such an approach can save us most of the overhead

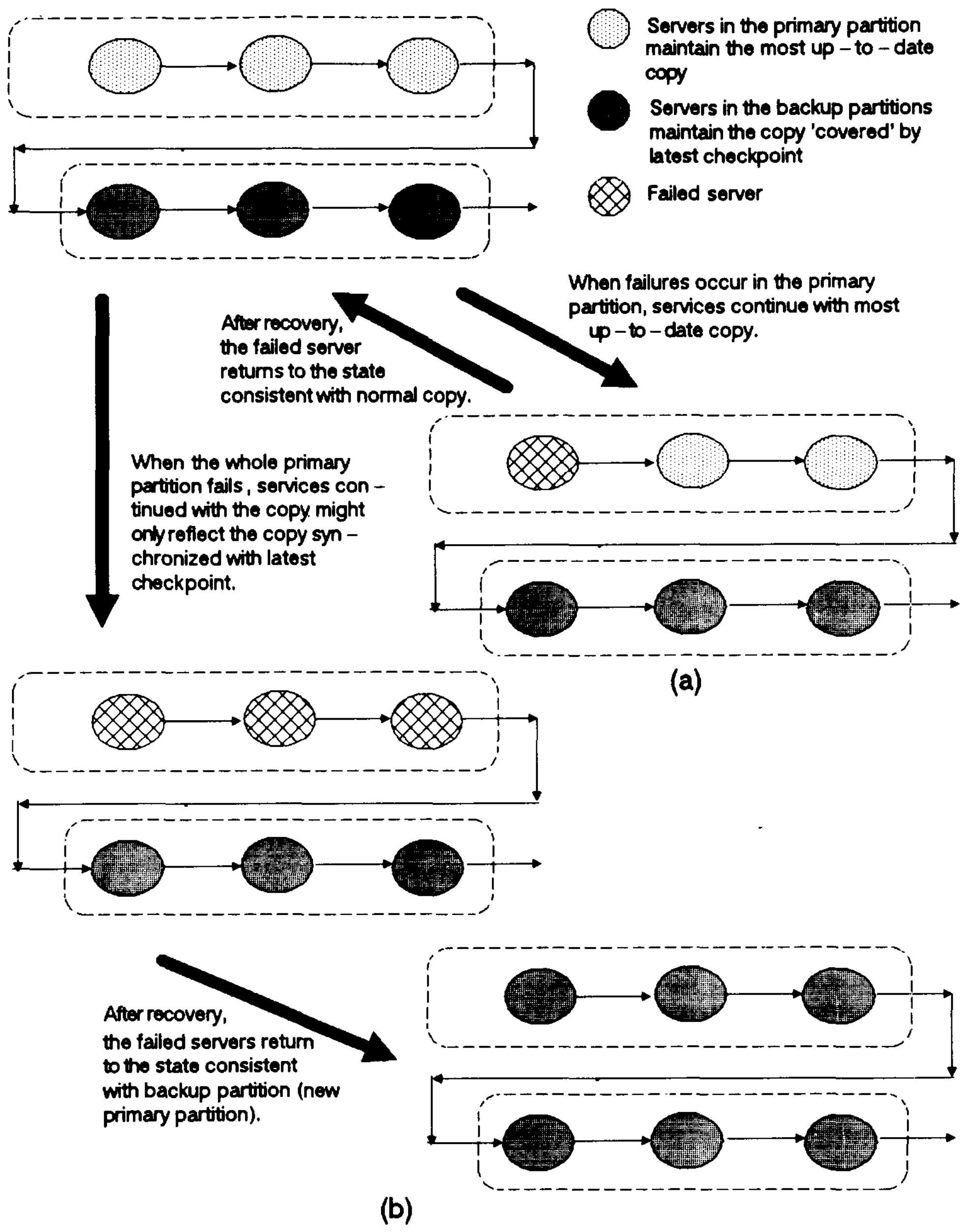

Servers in the primary partition maintain the most up – to – date copy
Servers in the backup partitions maintain the copy 'covered' by latest checkpoint
Failed server
When failures occur in the primary partition, services continue with most up – to – date copy.
After recovery, the failed server returns to the state consistent with normal copy.
When the whole primary partition fails, services con – tinued with the copy might only reflect the copy syn – chronized with latest checkpoint.
(a)
After recovery, the failed servers return to the state consistent with backup partition (new primary partition).
(b)

Figure 6. Recovery working schemes

of maintaining the mapping tables that is needed in other approaches. But whether it is worth while dedicating additional disks to guarantee fault tolerance remains questionable.

Our design is an eclectic one, which lies in the middle of the grain-size range. We call it directory-oriented replication. The unit of replication is the directory. When modifying a replicated file, the system will also search for and modify the replicas of the replicated file and preserve their consistency. We choose the directory as the unit of replication so that the mapping overhead can be reduced. The replica directory is just a place where we put the replicated files. It is not necessary to replicate all the files in the replicated directory; users can specify which files are to be replicated. The working scheme in the directory-oriented replication will be described in combination with name-space management in the following subsections.

Name-space management

A naming service, which binds global and high-level names to objects, is a key component in distributed file systems. A global naming service can provide names for objects in the system that can be passed between clients without change in interpretation, often referred to as absolute names.[24] In our distributed file system, we want to provide a global naming facility to meet the above requirements. More importantly, the transparency constraints must be satisfied. Users will access the distributed file system through our naming facilities as if they were using a single UNIX-like time-sharing system.

The *prefix* tables introduced by Welch and 0usterhout[25] have provided an efficient mechanism for locating files in the Sprite distributed system. The mechanism has four attractive features. First, clients negotiate directly with file servers to build the prefix tables, so we can avoid the need for a separate name server and make the file system more fault-tolerant. Secondly, their dynamic data structure permits the system to adapt gracefully for reconfiguration. Thirdly, by placing different entries in different clients' prefix tables, the mechanism can support private files and a simple form of replication. Finally, clients need not have local disks because they can use *network disks.*

The V's system's decentralized naming facility[24] is a similar mechanism with some variations. That is, the V mechanism replicates the top-level directories in the system. Such an approach will increase the reliability of top-level directories with the cost of maintaining consistency between copies of replicated directories. Another difference lies in the way of handling prefix cache misses (prefix tables are cached in the main memory). The V system will broadcast for the real servers, whereas Sprite leaves the naming tasks to the server that is nearest the root.

We apply the concept of prefix tables, and exploit their fault-tolerant potential. Some interesting features are added to the prefix tables for our model. In our description, a client is a machine that requests services and a (file) server is the machine that serves. A machine can be both a server and a client. A *domain* is a subtree that divides the file system. A *prefix is* the topmost directory in the domain. In our extended version of prefix tables, each entry is added with a new field, the back-up-table pointer. Thus each entry in a prefix table corresponds to one of the domains of the distributed file system. It contains the prefix of the domain, the name of the server that stores the domain, the token of the domain, and a pointer to the back-up table which stores the information of all the file server's replicas in a linear order. The client can send requests to the servers according to this linear order.

In the file system shown in Figure 7, the prefix table of one client will have the form shown in Figure 8. Each prefix (domain name) corresponds to a domain's root directory. A file server may implement more than one domain. Any naming service

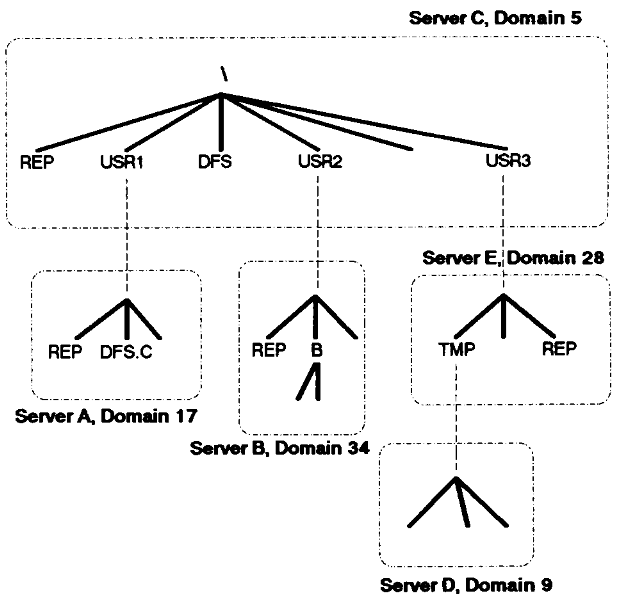
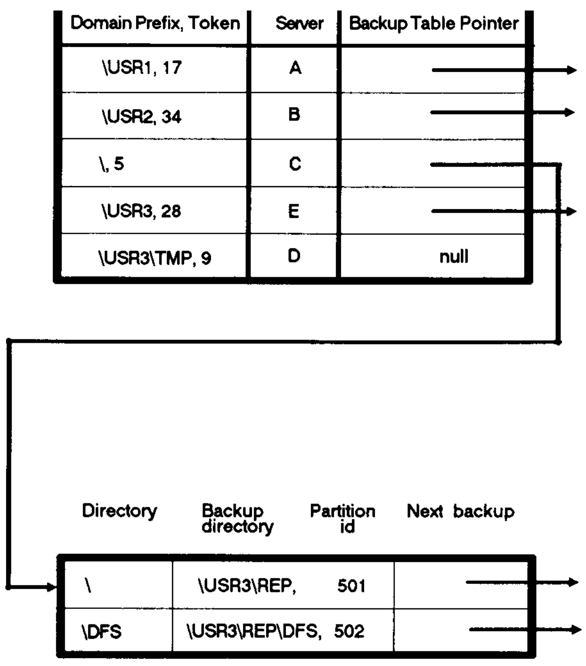


Figure 7. An example of file-system hierarchy. The root of server A is mounted on \USR1. *The root of server B is mounted on* \USR2. *The root of server D is mounted on* \USR3\TMP. *The root of server E is mounted on* \USR3. *The root of server C is* \. *The entire file system is a combination of servers A, B, C, D, E, but in the view of users it behaves as a single hierarchy*



Domain Prefix, Token	Server	Backup Table Pointer
\USR1, 17	A	→
\USR2, 34	B	→
\, 5	C	→
\USR3, 28	E	→
\USR3\TMP, 9	D	null

Directory	Backup directory	Partition id	Next backup
\	\USR3\REP,	501	→
\DFS	\USR3\REP\DFS,	502	→

Figure 8. A prefix table with its back-up tables

will be done by searching the table for matching prefixes. If several prefixes are found, the longest one is selected. Then the prefix is truncated from the path-name, and the remaining path-name is sent to the matching server for further search. For example, the path-name \USRI\DFS.C will match the prefixes \ and \USRI which correspond to domains 5 and 17 (server C and A), respectively. The latter is longer and is thus selected. \USRI is then truncated from the path-name, which becomes \DFS.C, and the new path-name is sent to server A for further processing. That is, server A is responsible for the remaining naming service. Such a mechanism can handle both absolute and relative path-names.

Now we turn to the replication aspect. If the file-name is prefixed by REP$, which indicates that it is important (needs replication), the 'search' operations are performed by using the original path-name, then the matched entry is also checked to see if there is any back-up. If so, the back-up table pointer field of the entry will point to the entry of the back-up table that gives the related information; otherwise it will be null. A back-up table for server C will be like that in Figure 8. In each entry a directory is associated with a back-up directory and a pointer to the next back-up entry, if any. This forms a replica list. To distinguish various partitions in the replica list, each back-up directory is associated with a partition id as shown in Figure 8. In our example there are two directories which offer backups: \, and \DFS. While at another client, the prefix table together with the back-up table may look different. From the above description, we can see that each client's view of the file system and reliability is defined by its prefix table and back-up table. The prefix is then replaced by the new prefix of the matching entry, and the new path-name is sent to the prefix table for another search. For example, consider the DOS command

```
COPY\AUTOEXEC. BAT \DFS\REP$I
```

The file-name prefix REP$ specifies that the file needs back-ups. The replica directory of \DFS is \USR3\REFIDFS. Hence two files \DFS\REP$I and \USR3\RERDFS\REP$I are created as shown in Figure 9. Similarly, when a user executes the DOS command

```
DEL REP$I
```

at current directory \DFS, both the files \DFS\REP$I and \USR3\REI+DFS\REP$I will be deleted.

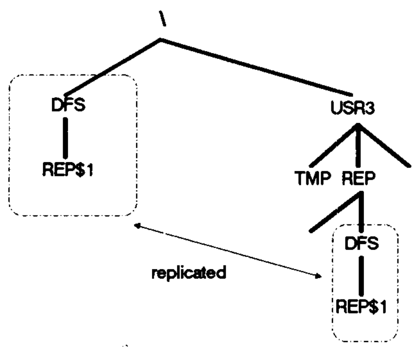


Figure 9. An example of file replication

However, what can we do if the directory is not replicated (no matching entry in the back-up table)? There are two kinds of strategies to make replication. One is to place the replica in the nearest parent directory. This approach can create many conflicts and cause more confusion. The other is to create the corresponding back-up directory automatically. This would lead to substantial overhead to maintain the back-up tables. We think that neither approach is appropriate. There is another alternative, which is to make the directory *replicable*. Users can specify whether they want to replicate the directory upon creation. Thus we leave the decisions to the users, and enhance the flexibility of the system configuration.

Command shell and DOS manager

Our model is implemented on an IBM PC-net with MS-DOS network environment as shown in Figure 10. The primary motivation for making such a decision is availability, because we already have a network of PC/ATs connected by PC-net, so we do not have to build another network from scratch. Another motivation for such a decision is the feasibility of implementation. Although MS-DOS resembles UNIX in many respects, there are still many differences between them. Most obviously, MS-DOS is a single-user (single-task) system. In a network environment, we need to handle requests for access to network resources. Fortunately, PC-net solves most of the problems. That is part of the reason why we have chosen the PC-net.

However, such an implementation decision has its cost. The more PC-net effectively imposes decisions on us, the less flexibility we can have. Therefore, we have still had to face some problems. The primary tasks for our implementation are:

1. *Integration:* despite its networking ability, PC-net is not a distributed system. We have to integrate all the disks in the network to provide a single, consistent UNIX-like file system view for the users. This is accomplished primarily by a command shell at the shell command level, called `dsh`.
2. *Fault tolerance: we* have to make sure that all the files that are important and need replication (prefixed by `REP$`) which are replicated, and all copies are consistent. In the case of a fault occurrence, continuous execution should be guaranteed. This is the job of the DOS manager at the system-calls level.
3. Recovery: when servers recover from errors, we should do all the recovery jobs. This is the task of the recovery manager.

The command shell serves as a simulation of a UNIX-like file system on a DOS

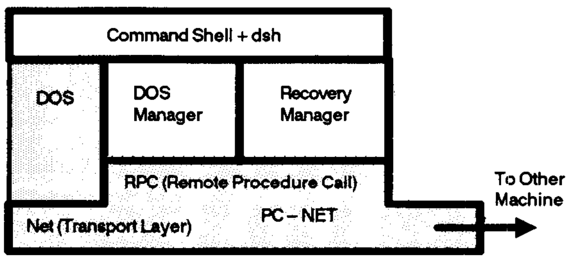


Figure 10. System architecture of our implementation

environment. Its primary job is to make the underlying PC-net with MS-DOS environment *transparent* to the users. All the UNIX-like path-names in the user's commands are translated to related DOS path-names. There can be no such concept as disk names or machine (PC) names. The users are dealing with an integrated, single and UNIX-like file system. For example, server A is mounted on the \USRI of server C. When the prompt is

[\USRI\DFS],

which indicates the current directory \USRI\DFS, this is actually server A's subdirectory \DFS. Through dsh, we create a virtual UNIX-like file system based on the PC-net with an MS-DOS environment.

In addition to path-name translation, the other basic function of dsh is to execute user programs. Our command shell can execute two categories of commands:

1. MS-DOS internal commands, such as COPY, DIR, REN, DEL, etc.
2. External commands (programs stored on the disk in the form as .EXE or .COM files).

Our dsh also supports environment variables, such as PATH. Thus the external commands can be executed if they are in the current directory or in the directories recorded in the PATH environment variable.

At the system-calls level, we need a set of new system services to meet our additional functional requirements. There are two ways to satisfy our need. One is to build a complete new kernel. The other is to add some facilities to DOS, i.e. to adjust DOS to fit our requirements. The former is too costly. Besides, one of our goals is to build an interface that will combine various file systems in the future, but not to build a new one. Therefore we have chosen the second approach.

Our DOS manager is an extension of ordinary DOS system calls. In addition to the ordinary services provided by DOS system calls, the DOS manager is also in charge of the file replication, fault detection and continuums execution. All file operations making requests to DOS will be redirected to the DOS manager first. If a reference is just to an ordinary file (not replicated), the request is passed to DOS directly; otherwise some preprocessing work will be done before the request is passed to DOS. After each normal DOS call returns, the DOS manager will check if there is any fault occurrence, and will ensure the smoothness of the execution if there is any copy available. Maintaining consistency between replicas is also the responsibility of the DOS manager.

Recovery management

The recovery we mention here aims primarily at the restoration of failed servers. Normally, the recovery-management structure consists of two components: the recovery manager and the log manager. [26] The recovery manager is responsible for managing the update operations. The log manager records all the operations done in the system. Since all file modifications are in-place updating, there is no cache, so the need for log manager's services is greatly reduced.

The recovery manager provides the following services:

1. The do operation, which commits and records the actions. Here the actions are equal to the file-related DOS calls.

2. The log operation, which records the failed servers and their back-ups' update operations after failures.
3. The duplicate operation, which copies the most up-to-date version files to the storage that is used to support restoration.
4. The restart operation, which reboots the system in order to bring the storage back to the committed state after a system crash.

All the operations described above should be *idempotent;* that is, repeated applications of any one of them should give the same result as a single application. For example, we can stop the duplicate operation at any time and repeat it from the beginning, while still retaining the result of a single execution.

IMPLEMENTATION

Command shell— dsh

Our dsh implementation consists of two layers. First, we should translate a user command's path-name into a path-name acceptable by DOS. Next, we should execute the command. In addition, in order to make the network transparent to the users, new versions of commands DIR, REN and DEL are provided. We have to do this because our experience shows that many DOS commands, such as DIR, REN and DEL, are still using FCB-style DOS calls, although MicroSoft Inc. has declared that they have stopped using these DOS calls after version 3.X. Since these FCB-style DOS calls are very hard to modify (see the next sub-section for explanation), we rewrite these commands by using the DOS system calls.

The translation mechanism is based on the extended prefix table. This table will reside in main memory as long as dsh is active. As for executing user commands, the MicroSoft C compiler that we have used offers a set of library functions, such as spawnle, spawnve and system, which can spawn child processes and execute DOS commands. For example,

```
system("TYPE  AUTOEXEC.BAT";
```

can execute the DOS command TYPE AUTOEXEC.BAT, and

```
spawnve(P_WAIT, "PE2.EXE", argument list, environment pointer);
```

can spawn a child process executing PE2.EXE. With these functions, implementing the command shell is a simple task.

DOS manager

Two things must be done by the DOS manager. First, if the file should be fault-tolerant (prefixed by REP$), the *redirector* should make sure that all copies of the file are up-to-date, and the DOS calls execute successfully. Secondly, if any copy of the file is not accessible (owing to errors such as server crashes), the *redirector* should record such failures and continue the execution provided that there is any copy that is accessible.

Table I The file-handle-related system calls

003EH	Close file
003FH	Read from tile handle
0040H	Write to file handle
0042H	Move the read-write pointer
0057H	Get/set file time and date

Table II The ASCIIZ-string-related system calls

003CH	Create a file and assign a handle
003DH	Open a file and assign a handle
0041H	Delete a file
4300H	Get file attributes
4301 H	Set file attributes
0056H	Rename a file
005BH	Create a new file (but do not destroy existed files)

In our implementation, the system calls that correspond to the file manipulations including the functions related to file handles and ASCIIZ strings are modified. They are listed in Table I and Table II, respectively. All these DOS calls are file-related. There are other calls that refer to files too. But most of them are used in networking or peripheral I/O, or system operations. At present we do not think there is any necessity to modify these calls.

We modify the DOS system calls through intercepting the interrupt 21H. As shown in Figure 11, we first replace the interrupt service routine with our program (*redirector*). Thus all the function calls will be redirected to appropriate routine for preprocessing if they are functions listed in the Tables. Then the normal DOS routine is followed by our post-processing routines. The jobs of replication, fault detection and forward progressing are all completed in this way.

As we have mentioned above, there are some DOS commands and even some old application programs still using FCB-style DOS calls, which have been allegedly discarded by MicroSoft Inc. At first we wanted to modify these DOS calls too, but

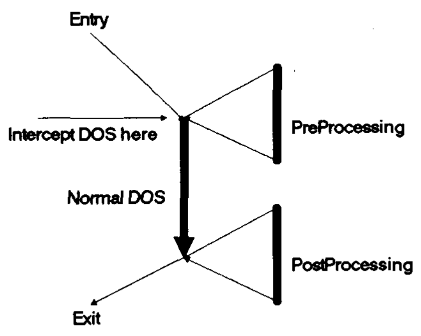


Figure 11. Intercepting INT 21H to modify DOS system calls

we discovered that it was almost impossible. The major problem comes from the naming structure of FCB. FCB'S file-name space is only 11 characters long (name + extension), which means that the file must be at its current working directory. To perform even a simple function, such as deleting a file, we must change the working directory first, then delete the file, and finally change the working directory back. To illustrate, both of the functions 13H and 41 H are file deletion. But the former is an FCB-style function. Thus deleting a file \DFS\SH.C when the current directory is \USR1 produces the following tasks: change the current directory to \DFS first, delete SH.C, and then change the directory back to \USR1. In an environment where all file-related calls need to be repeated many times (for the sake of replication), this will be quite an overhead. On the other hand, a single call, 41H, can complete all the tasks. As a consequence, we choose to abandon these DOS calls completely and replace them with compatible UNIX-style (handle) DOS calls.

Although only part of the DOS calls need to be modified, there are still substantial problems to be dealt with. The primary source of trouble is the fact that DOS calls are not *atomic.* One DOS call may affect the next DOS call or be affected by previous DOS calls that make the work of modification even harder. For example, DOS calls 4EH and 4FH are correlated and often used in directory listing. The former find the first file in the specified directory, and the rest of the files will be found by 4FH. We cannot get all the execution information from a single DOS call 4FH. That is the reason why we only guarantee fault-tolerant file accesses, not fault-tolerant DOS calls.

Moreover, many undocumented (or reserved) function numbers are actually being used (34H, 37H, to name but two). These lead to some strange behaviors in our implementation. That is why our implementation is not built completely at the system-calls level. Initially we wanted to implement the extended prefix table at system level. But merely intercepting every DOS call to modify the ASCIIZ strings does not lead to the desired results. Such characteristics of the DOS kernel bring us too many problems and have forced us to give up this idea.

DOS provides handle-style functions, which are similar to UNIX's counterparts, to access files. This is a very advanced programming concept. Since our DOS manager should replicate files according to a user's instruction, an extra mechanism is needed to manipulate replica handles. We have responded to this by devising handle mapping tables, in which every handle is associated with a handle list. As a new file is created, the returned handle and the handles of its replicas are filled in the next entry of the handle mapping tables. Thus, when handle functions are called, the DOS manager will search the table for the handle. If it is found, this means that the file is replicated. The same function will then be executed repeatedly with corresponding handles. When the function calls are ASCIIZ style, the back-up table is used to handle replication.

Recovery manager

In the current version, our recovery mechanism is simple. The do operation is completed by each DOS call. The restart operation is just to rebuild all the system tables and to reboot the system. The remaining operations are log and duplicate, which are based on the same mechanism, the error-log table.

In the error-log table, each entry records a failed server, its back-up servers, and a list of file names as shown in Figure 12. When the DOS manager detects a failed server, it will call the recovery manager to log such a failure. After that, as a file corresponding to the failed server is updated, the error-log table will append this file-name to the file-name list. When the failed server is restored, the duplicate routine will search the table for matching server id. Then all the files that have been updated during the failure will be copied from the back-up servers to the recovered server.

CONCLUSIONS

We have discussed the motivation, design and implementation of our distributed file system. Our goal of implementing a network-transparent, fault-tolerant (in the view of file access), distributed file system has been accomplished through the design and implementation of the command shell dsh, the DOS manager and the recovery manager. Besides the system implementation, we introduce a new model for file replication and the idea of directory-oriented replication. Furthermore, we have augmented the prefix-caching mechanism with replication facilities and have made the file system more fault-tolerant. We achieve fault tolerance by using replication. Our model combines the primary–stand-by and modular redundancy approaches to provide a more general and flexible replication. Our implementation is by no means optimal in all respects. It only provides another possibility for distributed file systems.

According to our implementation experience on the system, we have found that replication operations plus network overhead can slow down the response time profoundly. Intuitively, a replica must be accessed after a failed access to its original file. Hence the delay is at least twice as long as the delay for a normal access. However, the frequency of back-up usage depends on how reliable the servers and network are. There is still much work left to be done in this area. For example, the

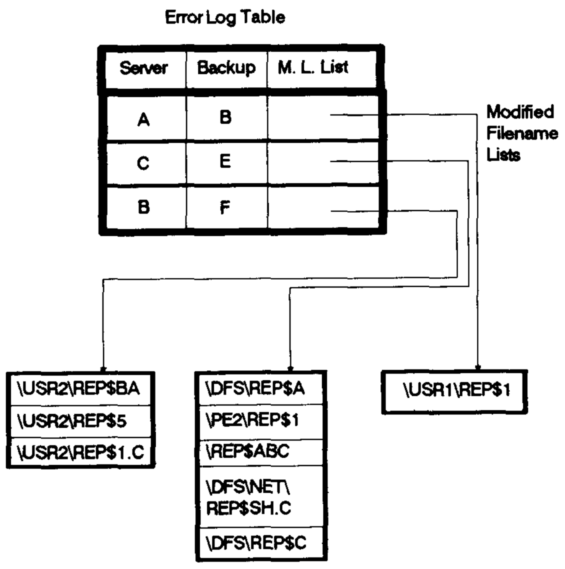


Figure 12. An error-log table with its modified file-name lists

replica list is predefine as a linear array of servers. A more flexible and efficient algorithm is needed to organize these servers and their back-ups. For achieving better performance, we need a formula to evaluate such parameters as how many back-ups are needed for a certain file, how many partitions, and how many replicas in a partition are most suitable for our system. These questions will be tackled in the near future.

In addition, we plan to construct a set of file-system services for various user applications. Furthermore, we intend to modify MS-DOS at the kernel level to enhance the transparency. In addition to the naming service we have provided, we want to introduce a more advanced memory-management scheme, such as caching, to improve our file-system performance. Hence we also need a more complex recovery algorithm. Another future direction is *heterogeneous distributed file systems.* If we consider the various types of machines existing in our laboratories, there is certainly enough motivation to combine them to form a heterogeneous computing environment.

ACKNOWLEDGEMENTS

The authors wish to thank the referees for their helpful suggestions and comments. They have improved both the content and clarity of this paper.

This work was supported by the National Science Council of the Republic of China under grant NSC78-0408-EO08-01.

REFERENCES

1. B. Walker, G. Popek, R. English, C. Kline and G. Thiel, 'The Locus distributed operating system', *9th ACM Symp. on Operating System Principles,* 1983, pp. 49–70.
2. D. R. Cheriton, 'The V distributed system', *Comm. ACM,* **31,** (3), 314–333 (1988).
3. J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B, B. Welch, 'The Sprite network operating system', *IEEE Computer,* (2), 23–26 (1988).
4. P. Dasgupta, R. J. LeBlanc Jr. and W. F. Appelbe, 'The Clouds distributed operating system: functional description, implementation details and related work', *11th IEEE Conf. on Distributed Computing Systems,* 1988, pp. 2–9.
5. J. S. Banino, J. C. Fabre, M. Guillemot, G. Morisset and M. Rozier, 'Some fault-tolerant aspects of the chorus distributed system', *5th IEEE Conf. on Distributed Computing Systems, 1985,* pp. 430–437.
6. J. E. Allchin and M. S. McKendry, 'Synchronization and recovery of actions', *2nd ACM Symp. on Principle of Distributed Computing,* 1983, pp. 31–44.
7. D. R. Brownbridge, L. F. Marshall and B. Randell, 'The Newcastle connection or UNIXes of the world unite!', *Software–Practice and Experience,* **12,** (7), 1147-1162 (1982).
8. T. D. M. Purdin, R. D. Schlichting and G. R. Andrews, 'A file replication facility for Berkeley Unix', *Software–Practice and Experience,* **17,** (2), 932–940 (1987).
9. G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, 'The Eden system: a technical review', *IEEE Trans. Soft. Eng.,* **SE-11,** (1), 43–59 (1985).
10. O. P. Brereton, 'Management of replicated files in a UNIX environment', *Software—Practice and Experience,* **16,** (8), 771–780 (1986).
11. P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987.
12. K. S. Yap, P. Jalote and S. Tripathi, 'Fault tolerant remote procedure call', *11th IEEE Conf. on Distributed Computing Systems,* 1988, pp. 48–54.
13. J. J. Bloch, D. S. Daniels and A. Z. Spector, 'A weighted voting algorithm for replicated directories', *JACM,* **34,** (4), 859–909 (1987).
14. J. Bartlett, 'A NonStop$^{\text{tm}}$ kernel', *8th ACM Symp. on Operating System Principles,* 1981, pp. 22–29.
15. K. P. Birman, T. A. Joseph, T. Raeuchle and A. E. Abbadi, 'Implementing fault-tolerant

distributed objects', *4th ACM Symp. on Reliability in Distributed Software and Database,* 1984, pp. 124–133.

16. A. Borg, J. Baumbach and S. Glazer, 'A message system supporting fault tolerance', *9th ACM Symp. on Operating System Principles,* **17,** (5), 90–99 (1983).
17. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel, 'A network transparent, high reliability distributed system', *8th ACM Symp. on Operating System Principles,* 1981, pp. 169–177.
18. P. A. Bernstein and N. Goodman, 'An algorithm for concurrency control and recovery in replicated distributed databases', *ACM Trans. Database Systems,* **9,** (4), 596–615 (1984).
19. D. K. Gifford, 'Weighted voting for replicated data', *7th ACM Symp. on Operating System Principles,* 1979, pp. 150–162.
20. C. J. Date, *An Introduction to Database Systems, Vol. I, Fourth Edition,* Addison-Wesley Publishing Company, 1986.
21. T. Anderson and P. A. Lee, *Fault Tolerance-Principles and Practice,* Prentice-Hall Inc., 1981.
22. M. Maekawa, A. E. Oldehoeft and R. R. Oldehoeft, *Operating Systems—Advanced Concepts,* Benjamin/Cummings Publishing Inc., 1987.
23. I. L. Traiger, J. Gray, C. A. Galtieri and B. G. Lindsay, 'Transactions and consistency in distributed database systems', *ACM Trans. Database Systems,* **7,** (3), 323–342 (1982).
24. D. R. Cheriton and T. P. Mann, 'A decentralized naming facility', *Tech. Rep. STAN-CS-86-1098,* Computer Science Dep., Standford University, 1986.
25. B. Welch and J. Ousterhout, 'Prefix tables: a simple mechanism for locating files in a distributed system', *9th IEEE Conf. on Distributed Systems,* 1986, pp. 184–189.
26. J. N. Gray, 'Notes on database-operating system', *Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60,* Springer-Verlag, N. Y., 1978, pp. 393–481.