Proc. Natl. Sci. Counc. ROC(A)
Vol. 23, No. 3, 1999. pp. 402-410

(Scientific Note)

A Fault-Tolerant Model for Replication in Distributed-File Systems

TZUNG-SHI CHEN*, CHIH-YUNG CHANG**, JANG-PING SHEU***, AND GWO-JOHN YU***

*Dept. of Information Management
Chang Jung University
Tainan, Taiwan, R.O.C.
**Dept. of Information Science
Tamsui Oxford University College
Taipei, Taiwan, R.O.C.
***Dept. of Computer Science and Information Engineering
National Central University
Chungli, Taiwan, R.O.C.

(Received April 20, 1998; Accepted September 1, 1998)

ABSTRACT

In this paper, we propose a new fault-tolerant model for replication in distributed-file systems. We combine the advantages of the modular redundancy, primary-stand-by, and weighted priority schemes to address the fault-tolerant model. To fulfill the fault-tolerant requirements, we use the ideas of directory-oriented replication and the extended prefix table while incorporating a two-level heap data structure. In a consideration of practical circumstances, the heap, using an array instead of a linked-list tree structure, has a dynamically adjustable property, which can be used to easily modify the tree structure in the distributed-file system. Therefore, this fault-tolerant model provides low-cost overhead, a variety of transparency capacities, a fault-tolerant capacity, and a parallel commitment capacity on backup replicas for distributed-file systems.

Key Words: distributed-file systems, fault tolerance, network transparency, recovery, replication

I. Introduction

Over the last decade, distributed processing has become more and more important and attractive with progress in the areas of computer networks and distributed systems. For file accesses, current centralized-file systems unfortunately limit performance and availability because all accesses with read or write operations need to go through the central file server. Therefore, for distributed file access, many researches have focused on the design and implementation of distributed-file systems (Anderson *et al.*, 1996; Devarakonda *et al.*, 1996; Kistler and Satyanarayanan, 1992; Howard *et al.*, 1988; Satyanarayanan, 1990; Walker *et al.*, 1983). The storage media distributed over such networks may potentially incur some unexpected faults. Because of hardware or software failure in distributed-file systems, these systems have to provide a fault-tolerant capability so as to tolerate faults and to try to recover from these faults.

A lot of researches have aimed to develop distributed-file systems with fault-tolerant capacity (Banino *et al.*, 1985; Cheng and Sheu, 1991; Dasgupta *et al.*, 1988; Kistler and Satyanarayanan, 1992; Purdin *et al.*, 1987; Satyanarayanan, 1990). They have developed replication and Redundant Arrays of Inexpensive Disk (RAID) (Chen *et al.*, 1994) techniques to provide redundancy for fault tolerance. Traditionally, distributed-file systems have relied on redundancy for high availability. In general, file systems replicate at the server-level, directory-level, or file-level to deal with processor, disk, or network failures. Redundancy allows these systems to operate easily and continuously despite partial failure at the cost of maintaining replicas (copies) in the file system. Replicas provide users with a fault-tolerant environment so that they need not be concerned with where the replica being accessed actually resides. Thus, the main purpose of this paper is to provide fault tolerance for distributed-file systems by using the redundant technique with lower-cost overhead of maintaining faults and by allowing a parallel commits capacity to reduce the turnaround time of

service requests issued by clients.

We focus in this paper on the model construction of a distributed-file system with fault tolerance. In our fault-tolerant model, forward progress has to be guaranteed. If any replica is available when an unexpected fault occurs, execution should proceed continuously and as smoothly as possible. The most popular approach we use is to periodically set checkpoints on the file system. When the failed server resumes after a failure, the stored data may not be the most up-to-date data. The data should then be corrected according to the information obtained from the checkpoints the system has set up. To fulfill the fault-tolerant requirements, we use the ideas of directory-oriented replication and the extended prefix table (Cheng and Sheu, 1991) while incorporating a two-level heap data structure. In a consideration of practical circumstances, the heap, using an array instead of a linked-list tree structure, has a dynamically adjustable property which can be used to easily modify the tree structure in the distributed-file system. Therefore, this fault-tolerant model provides low-cost overhead, a variety of transparency capacities, a fault-tolerant capacity, and a parallel commitment capacity on backup replicas for distributed-file systems.

The rest of this paper is organized as follows. Section II introduces our model and related work. In Section III, we propose our fault-tolerant model for replications in distributed-file systems to improve efficiency over our previous work (Cheng and Sheu, 1991). From the practical point of view, we derive some data structures and operations to design our proposed fault-tolerant model in Section IV. Finally, conclusions are summarized in Section V.

II. Related Work

It is important to deal with hardware failures, such as server crashes, in a distributed-file system. We will concentrate our discussion on hardware failures and on fault tolerance by means of replication. In the literature (Bloch *et al*., 1987; Guerraoui and Schiper, 1997), the use of replication to provide the capacity of fault tolerance as in the distributed schemes can be classified into three approaches: the *primary-stand-by* approach, the *modular redundancy* approach, and the *weighted voting* approach.

For the sake of tolerating faults, some researchers have focused their research on replica mechanism design. The primary-stand-by approach (Guerraoui and Schiper, 1997) first selects one copy as the primary one while the others are stand-by ones. If one copy is a primary one, all of subsequent requests from clients are sent to it only. The stand-by replicas only synchronize with the primary copy periodically while they are not available to service requests. When failures occur in the primary copy, one of the stand-by replicas will be chosen as the new primary one. Then, the request service will continue to go on from the most recent commit point. The second one is the modular redundancy approach (Banino *et al*., 1985) which provides the system with no distinction between the primary copy and stand-by ones. The requests are sent to all of the backups simultaneously; that is, the service is performed on all of machines for each copy. Thus, there exists at least one correct copy in these machines to achieve the fault tolerance. In the third approach, weighted voting (Gifford, 1979), all replicas of a file are assigned a certain number (weight) of votes. We call these replicas representatives. Any request, including a read or write operation, is performed on a set of representatives called a request quorum. Any read (write) quorum which has a majority of the total votes of all the representatives is allowed to perform the corresponding operation.

In our previous work (Cheng and Sheu, 1991), we presented a scheme which blends together the primary-stand-by approach and the modular redundancy approach. We divided all of the backups into several partitions connected to a linear form. The first partition was called the primary partition and the others were called backup partitions. In order to improve system performance over the previous one, we will propose a new fault-tolerant model for replication in distributed-file systems in the next section.

III. A Fault-Tolerant Model for Replication

1. A Fault-Tolerant Model

The basic concept of our proposed fault-tolerant model is described as follows. We combine the advantages of the modular redundancy, primary-stand-by, and weighted priority schemes to create a new fault-tolerant model. We initially divide all of the replicas for fault tolerance into several groups with variable size depending on the practical situation. Each group is assigned a priority corresponding to its weight and depending on its significance. The larger the priority value, the more significant it is. Each group acts as a modular redundancy unit. We call the group with the highest priority the *primary group*, and the others are called *backup groups*. Requests are sent to all backups in the primary group for service in parallel. This is the same as in the primary-stand-by case. Therefore, we can enhance flexibility to instantaneously forward progress to the other groups after completing

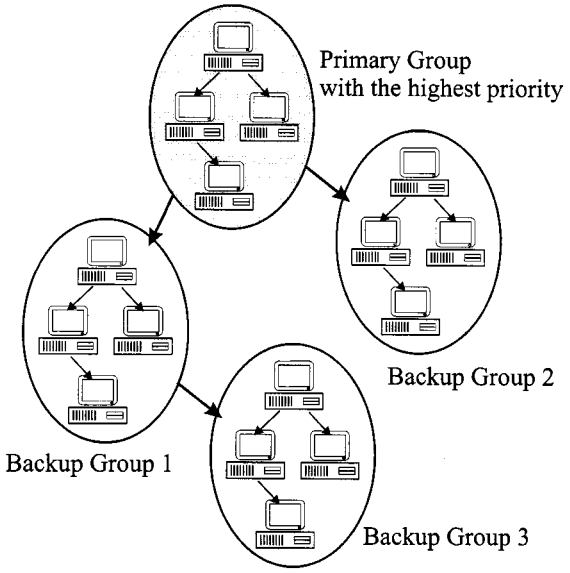


Fig. 1. Replica groups with a two-level heap structure in our model.

the processing commits of the primary group. For each replica, we can choose the significant servers in the primary group or backup groups, which fully depend on some performance criteria, such as server loads, proximity constraints with spatial locality, server computing power, and so forth. This kind of selection is similar to the Accessible Volume Storage Group (AVSG) on the Coda system (Satyanarayanan, 1990).

A heap structure, which is a binary tree, is used to connect these groups by means of group priority. In the heap, the weight of a node is less than or equal to that of its parent node. The root of this heap tree has the highest priority with the largest weight value. We also use the same technique to construct the heap structure for all of the replicas in each group if each replica in a group also has a designated priority. In this way, we can speed up the commits of client requests easily by applying a parallelizing technique. Thus, a client is no longer responsible for dealing with all calls and returning responses. Moreover, performing parallel commits for each client request is not only done for each group, but also among groups to achieve high-performance computing in a distributed environment.

As mentioned above, we demonstrate our model using an example shown in Fig. 1. Assume that there are sixteen replicas (servers) for a certain data object which are divided into four groups, the first one called a primary group and the others called backup groups. These four groups are connected into a heap tree, called an *inter-group heap*, based on the designated priority of each group. All of the replicas in each group are also connected into a heap tree, called an *intra-group heap*. We assume that all the replicas in one group have the same priority. The replicas could be assigned different levels of priority, depending on the practical application.

When a client issues a request, the service request is sent to the root of an intra-group heap which is the root of the inter-group heap. If this request is an update operation, the first replica must propagate the request to the replicas in the primary group. Then, all of the other requests submitted by clients need to be queued to wait for processing later. In order to maintain consistency, the requests in the queue are continuously delivered to the other groups in parallel until a checkpoint is reached. If no failure occurs, the replicas in the primary group will get the request. If this request needs to update the content of the data, the request has to be propagated to the other groups in parallel. We will consider an example shown in Fig. 2 with four servers in the primary group. The client issues a request for an update operation to the distributed-file system. Server 0 gets the request at time 1 and then passes the request to its left child, server 1, at time 2. At time 3, servers 0 and 1 pass the request to the respective server 2 (the right child of server 0) and server 3 (the left child of server 1) in parallel. In the reverse order, these servers reply a result message to the client. After relaying the response to the client, server 0 (the root of the heap) goes on propagating the service request to the backup groups by means of the inter-group heap. The numbers between the two servers indicate the message sequence of a request-service cycle. Therefore, we allow parallel commits with group communication to reduce the turnaround time of service requests issued by the client in the intra-group heap.

All replicas (servers) in the primary group will relay the request and wait for the result (response) of its child in the inter-group heap. Replicas in the backup groups need not synchronize with replicas in the pri-

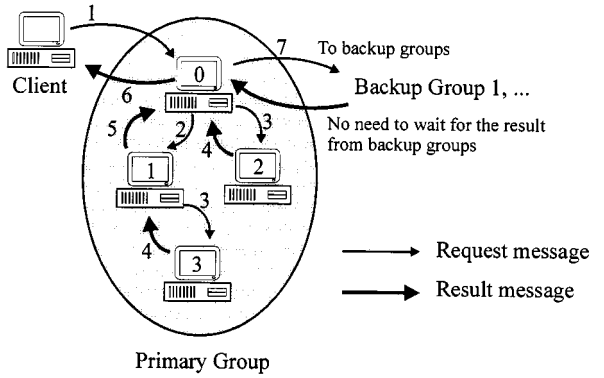


Fig. 2. Request and result messages are sent in parallel if no server has failed.

mary groups. Among groups of the inter-group heap, information propagation with request and reply messages only takes place on root servers of intra-group heaps. These actions and related operations for parallel commits can be easily examined in Fig. 2.

Now, we will discuss the time complexity of performing a request issued by a client and getting a response from this distributed-file system if no server fails in this system. Assume that the average service time for a certain server to complete execution of a request (an update operation) is T_s. The average communication time between two servers in the network is assumed to be T_c. The number of backup groups is N_g. Thus, the height of the inter-group heap is $\lfloor \log N_g \rfloor + 1$. The number of servers in the backup group k is G_k, and the height of its corresponding intra-group heap is $\lfloor \log G_k \rfloor + 1$, where $0 \le k \le N_g - 1$. Hence, it takes the maximum number of transmission steps, $2\lfloor \log G_k \rfloor$ steps, to deliver a message from the root to a leaf server in group k along the intra-group heap. Thus, the time needed to complete execution of a request-service cycle in group k is $O(\log G_k(T_s)+4\log G_k(T_c))=O(\log G_k(T_s+T_c))$, which consists of the time for relaying the request, executing the update operation, and gathering a response. Hence, the total time complexity of performing the completion request-service cycle on the two-level heap structure is

$$O(\log N_g(\log(\max_{0 \le k \le N_g - 1} G_k)(T_s+T_c))).$$

Because of the nice property of the two-level heap, we are able to efficiently and quickly complete execution of a request for a client.

Based on the above construction, there is a problem of how to guarantee and enforce the single-copy semantics in this system. In our fault-tolerant model, we use a useful data structure, a heap, incorporated with a hybrid structure, i.e., a two-level heap structure. In each intra-group heap, the single-copy will be preserved and guaranteed by propagating and performing the update operation in parallel. The request issued by the client is then relayed through the inter-group heap. By means of the structure with two layers, the communication overhead is efficiently distributed over all the replicas, and the single-copy semantic is still remained.

The fault-tolerant model we have addressed can provide several advantages for tolerating faults and is applicable to distributed environments. First, this model has inherent support for wide-scale replication. It also provides a high degree of replication flexibility by precisely and dynamically specifying the replica data. It has the nice property of being easy to handle so as to maintain the data structures for connecting the replicas and to manipulate these by means of dynamically adjustable operations. Therefore, this improved fault-tolerant model definitely meets the requirement for highly scalable and available distributed-file systems.

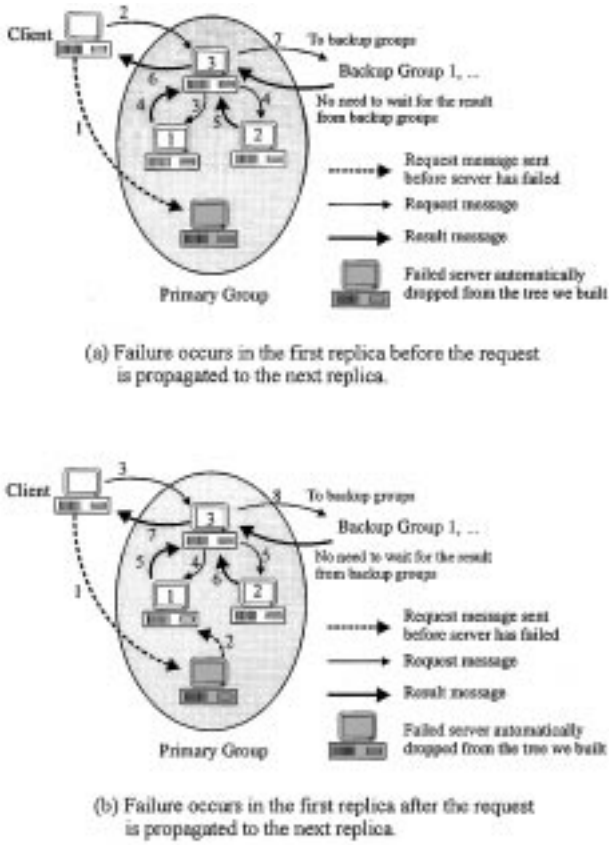


Fig. 3. Failure occurs in the primary group.

2. Server Failures

In a distributed-file system, there may exist some unexpected faults on each replica. Assume that some failures occur in the primary group. There are two failure-event times: before or after the replica server propagates its request. We will first illustrate the basic concept with an example shown in Fig. 3, where we show how it works when a failure occurs in the primary group. Here, all of operations, including adjusting operations for heap and message transmission for sending a request or acknowledgement are systematic and automatic in this fault-tolerant model. In the case where server 0 has failed, the failed server is automatically removed from the intra-group heap; then, the request is relayed to the other servers. As shown in Fig. 3(a), the client issues a request to server 0 failed to work well. As a result, the intra-group heap can be immediately reconstructed into another intra-group

heap, where the node of server 0 is dropped from the heap; then, the client reissues the same request to server 3 in step 2. That is, we interchange the faulty node, to be removed, with the last node and then percolate the changed node down or up the modified heap. As shown in Fig. 3(b), the client issues a request to server 0. While the server 0 propagates the request to the child of the intra-group heap, a failure occurs and is detected in server 0. As a result, the intra-group heap can be immediately reconstructed into another intra-group heap, where the node of server 0 is dropped from the heap; then, the client reissues the same request to server 3 in step 3. As depicted in Figs. 3(a) and (b), the issued request is then propagated in parallel to the replicas on the reconstructed intra-group heap.

As mentioned above, we will discuss the cases of node failures in a group in detail. The corresponding intra-group heap is denoted by H_{tree}. Let $W(n)$ denote the value of priority (weight) of a node n. Assume that node n_i fails after or before the request is propagated to the next node. We denote $Par(n_i)$ as the parent of node n_i in H_{tree}. When the request was relayed to $Par(n_i)$, we guarantee that the ancestors of node n_i have served the service request, and that the descendants have not received the service request. Also assume that there exist two children, the left child $L(n_i)$ and the right child $R(n_i)$, of n_i. We will remove the faulty node n_i from H_{tree} and interchange n_i with the last node n_j in H_{tree}. Based on the situation of request propagation shown in Fig. 4, there exist two circumstances while server n_i fails in H_{tree}.

In the first circumstance, we assume that the faulty node n_i is the ancestor of the last node n_j as depicted in Fig. 4(a).

Case 1: The faulty node n_i is removed from H_{tree}, and we replace the position of n_i with the last node n_j to form another tree H'_{tree}. It is a well-known fact that $W(n_j) \le W(L(n_i))$ and $W(n_j) \le W(R(n_i))$. The node n_j is percolated down H'_{tree} to a proper position to generate a reconstructed heap. After that, we deal with the two cases of failure-event shown in Fig. 3.

Secondly, we assume that the faulty node n_i is not the ancestor of the last node n_j as depicted in Fig. 4(b) with the following two cases.

Case 2: The faulty node n_i is removed from H_{tree}, and we replace the position of n_i with the last node n_j to form another tree H'_{tree}. Suppose node n_j has not received the service request. There are two sub-cases described below.
(1) Suppose $W(n_j) < W(L(n_i))$ or $W(n_j) < W(R(n_i))$. Node n_j is percolated down H'_{tree} to a proper position to generate a reconstructed heap. After that, we deal with the two cases of failure-event shown in Fig. 3.
(2) Suppose $W(n_j) \ge W(L(n_i))$ and $W(n_j) \ge W(R(n_i))$. $Par(n_i)$ first relays the request to n_j. Then, node n_j is percolated up H'_{tree} to generate a reconstructed heap. Now, we assume that n_k with the request information occupies the original position of n_i in H_{tree}. Then, n_k proceeds to relay the request to its children and descendants.

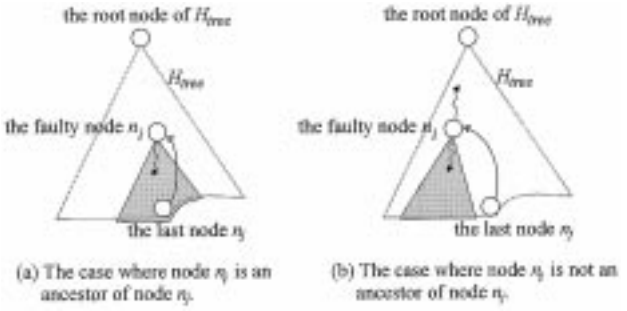


Fig. 4. The cases of server failures in the intra-group heap H_{tree}.

Case 3: As in Case 2, suppose node n_j has received the service request. Thus, node n_j, which does not need to be serviced twice, would reply with the response only. It will be percolated down or up H'_{tree} according to its priority. Then, n_k, as in Case 2, proceeds to relay the request to its children and descendants.

We assume that the operation for dealing with the above cases is atomic and indivisible in our proposed model; i.e., there are no two or more operations which interfere with each other. From the above cases, the operation of the node n_j being percolated up and down the heap only affects its ancestors and descendants, respectively. This is because we use the nice adjustable property of the heap structure.

Until now, we have only considered the case of failures occurring in a group. Through the dynamically adjustable property of the heap, we can efficiently remove faulty servers from an intra-group heap or even from the entire group in an inter-group heap.

3. Failure Recovery

In this approach, we use a simple recovery scheme, *backward recovery*, which stores all objects before the atomic action begins. Reconsider the example shown in Fig. 1. If there exist some faults, the working

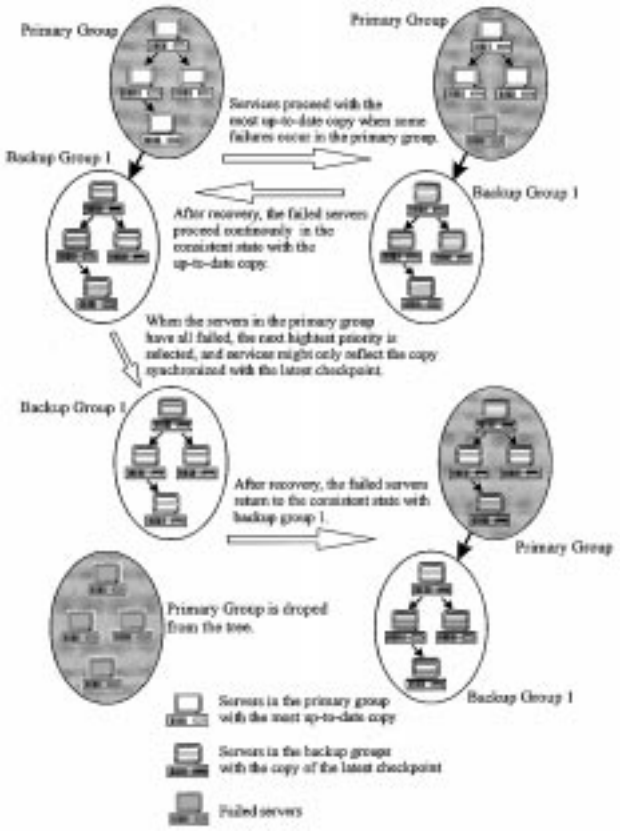


Fig. 5. The scheme for failure recovery.

recovery scheme of our fault-tolerant model is that shown in Fig. 5. When some failures occur in the primary group, services on the file system proceed with the most up-to-date data copy. Once the failed servers in the primary group are recovered, we will immediately restore the modified data to a consistent state as long as there exist no updates in the current group. If the entire primary group was failed, the data in the backup groups may be obsolete because these replicas in backup groups do not have the latest updated data. However, the next most significant group with the second highest priority takes over the primary status and is selected as the primary group while the original primary one waits for recovery. The services on the file system may reflect synchronization of the data copy with the latest checkpoint which was set up. Once the failed servers in the primary group are recovered, their current states are updated so as to be consistent with those of the servers in the new primary group, backup group 1.

When a faulty server has recovered, we insert the corresponding node into the intra-group heap. We first append the recovered node to the last position in the intra-group heap. It is then percolated up the intra-group heap to the correct position according to its priority.

As mentioned above, we only consider how to recover faulty servers in the primary group. If failures occur in other backup groups, we can also apply the above scheme to recover from the failures. Therefore, through the dynamically adjustable property on heap, our model can efficiently reconfigure and work well.

IV. Extended Prefix Table Structures

The key goal of our proposed distributed-file system is to provide users with transparent and fault-tolerant data access. An efficient way to locate and access files is to use the extended prefix table to design a distributed-file system (Cheng and Sheu, 1991). In our proposed fault-tolerant model, there exist automatic abilities of parallel commits for each service-request and fault recovery. Therefore, this system, through the mechanism we have designed and its manipulation operations, provides this ability with four types of transparency: location transparency, replication transparency, concurrency transparency, and failure transparency. In this section, we will describe the data structures of the extended prefix table for tolerating unexpected faults in the distributed-file system.

1. Directory-Oriented Replication

In a fault-tolerant system via replication scheme, the replication granularities of a data object vary in size from a file to an entire disk. For larger or smaller granularity as one replica unit, there much space is needed to maintain the mapping tables (Cheng and Sheu, 1991). Therefore, we focus on using directory-level replication, the middle level of granularity, to design our distributed-file system. That is, we replicate the data files in a designated directory, without including the data in its subdirectories, as a replication unit.

We will first explain the concept while exploring its fault-tolerant potential and ability. A client is a machine that requests services, and a server is also a machine that serves each client. A *domain* is a subtree that is part of the file system. We integrate and mount these domains together to produce a distributed-file system. In Fig. 6, we show an example of a file system with four servers. Each server has a unique domain number to distinguish it. For each domain, we create a REP subdirectory, if necessary, in which to store the replication data as a replica for fault tolerance. Within each REP subdirectory on each server, we create subdirectories whose names are domain numbers as shown in Fig. 6. In Fig. 6, there are three subdirectories, “5”, “17”, and “27”, within REP on server *A*. We create

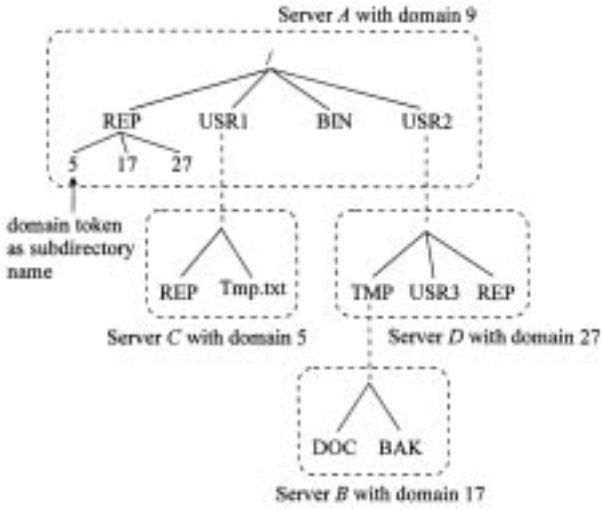


Fig. 6. An example showing the distributed-file system hierarchy.

these subdirectories in REP not only so that this file system can easily handle the replicas for fault tolerance, but also, because they can keep the file system from accessing incorrect data replicas. This is because it is possible that, for two files, there exists the same file name with different data content within the same subdirectory for two servers.

2. Data Structures

We have used and extended the *prefix table* model (Welch and Ousterhout, 1986) to design our distributed-file system in order to efficiently locate and access files for convenience. We will first describe the data structures implemented in our distributed-file systems. A *prefix* is the topmost directory in the domain. In Fig. 7, following the above example, we show an example to demonstrate the concept of the extended prefix table with its backup and group tables. Each entry within a prefix table corresponds to one of the domains in the file system. In the prefix table, to each entry is added a new field, i.e., the backup table pointer we extended. For example, there is a backup table pointer within the entry for server *A*. This pointer points to a backup table corresponding to server *A* with domain 9. That is, the two directories, "/" and "/BIN" within server *A*, have been replicated automatically or manually by the system. The directory "/" is designated for some group recorded to the group table with only one entry with weight 10. By keeping track of the pointers depicted in Fig. 7, we can store the two replicas of the data files in the directory "/" on server *A* into the directories "/USER1/REP/9/" and "/USER2/REP/9/" on the domain 5 (server *C*) and domain 27 (server *D*), respectively.

Now, we will introduce three data structures written in the C language as shown in Fig. 8 in order to express the above ideas. Using these data structures, Group, Backup, and BackupEntry, we can easily represent the above structure of the extended prefix table. In the GroupTable, we use an array to represent a heap structure, called an inter-group heap. This is because, using an array, we can easily employ these operations to adjust the heap tree. By means of the same construc-

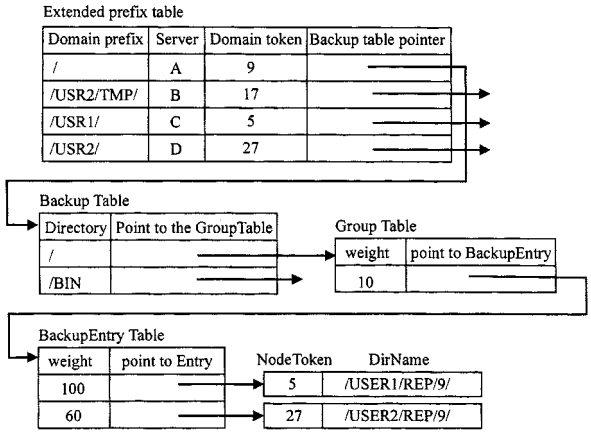


Fig. 7. An extended prefix table with its backup tables and group tables.

```
struct Group
{       int weight ;
        struct Backup *pointer ;
} GroupTable[ NoGroup ] ;
```

(a) The data structure of Group.

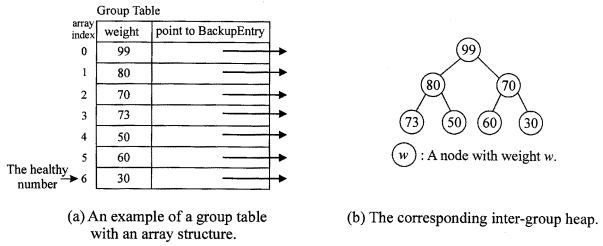
```
struct Backup
{       int weight ;
        struct BackupEntry *entry[ NoEntry ] ;
} ;
```

(b) The data structure of Backup.

```
struct BackupEntry
{       int NodeToken ;
        char *DirName ;
} ;
```

(c) The data structure of BackupEntry.

Fig. 8. The data structures for replica backup.



(a) An example of a group table with an array structure.

(b) The corresponding inter-group heap.

Fig. 9. A group table with heap structure representation of a given example.

tion, we can also use an array to represent the BackupEntry table, called an intra-group heap. Within the hybrid data structure, it is clear and easy to form a two-level heap structure.

3. Fault Tolerance

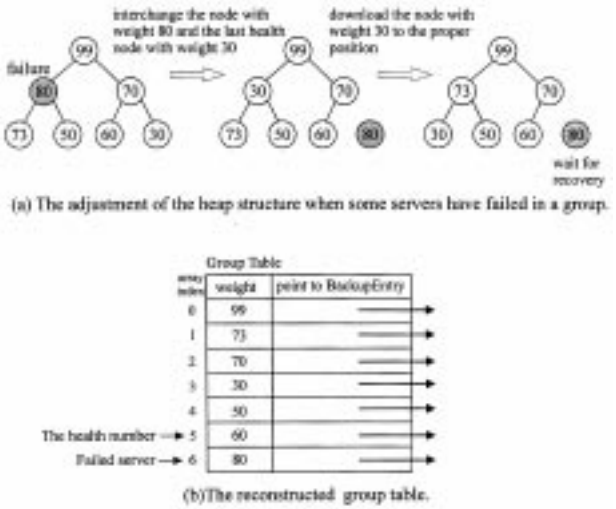
Based on our proposed model and the data structures, we will demonstrate how to provide the fault-tolerance ability when a fault is detected and to then recover to a consistent state. In our proposed model, we can view this structure as a two-level hierarchical structure: one level is the intra-group heap, and the other is the inter-group heap. For both kinds of heap trees, we use the same operations to adjust changes in the trees, such as when one fault has been detected (removed from the heap) and is waiting for recovery or when a fault has been recovered (added to the heap).

Now, we will only consider an example of when an entire group fails as shown in Fig. 9. Assume that there are seven groups used to maintain a specific directory, and that their corresponding weights are those shown in Fig. 9(a). Thus, the indicator of the health number, which after adding one is the number of groups in which there exist some non-faulty replicas, points to the number 6; i.e., there are seven healthy groups. The inter-group heap is shown in Fig. 9(b), where each node denotes a group with its weight inside the circle. Group 0 with weight 99 is referred to as the primary group; the others are referred to as backup groups.

Assume that backup group 1 with weight 80 has been detected to have failed, i.e., the replicas within it need to wait for recovery. Now, we have to drop the faulty group with weight 80 from the inter-group heap as shown in Fig. 9(b) and reconstruct the modified inter-group heap with six groups as shown in Fig. 10(a). The time needed to perform the drop and reconstruction operations while a node is percolated down the heap is in practice dependent on the height of the inter-group heap, assuming $H_t = \lfloor \log N_g \rfloor + 1$, where N_g is the number of backup groups. Hence, the time is $O(H_t)$. Then, the dropped group needs to wait for recovery. The corresponding reconstructed group table is shown in Fig. 10(b). Once all of the failure servers within the group have recovered, we can restructure the inter-group heap in the original heap style. The time needed to perform the insertion operation while a node is percolated up the heap is also dependent on the height of the inter-group heap. Thus, the time is also $O(H_t)$. Therefore, less overhead is needed to efficiently handle all of the data structures for replication and manipulation operations using our proposed model.

V. Conclusions

In this paper, we have proposed a new fault-tolerant model for distributed-file systems. We have extended the previously proposed fault-tolerant model to create this distributed-file system model. The proposed model provides low overhead as well as transparency, fault tolerance, and a parallel commitment capacity for distributed-file systems. Using this model, we can efficiently manipulate any update of data structures for replication on distributed-file systems. From the point of view of scale, our addressed model can be easily applied to local-area networks (LANs) and wide-area networks (WANs). Communication among these backup servers in WANs is significant greater than that in LANs. To reduce the communication cost, we can use the proposed two-level heap to construct and link these backup servers in a WAN in the following way. Each intra-group heap is constructed in its corresponding LAN. Then, we have the whole file server system via the inter-group heap in a WAN. The design and implementation of this fault-tolerant model,



(a) The adjustment of the heap structure when some servers have failed in a group.

(b) The reconstructed group table.

Fig. 10. A group table when all of the servers have failed in the group with weight 80.

incorporated into the previous one is under investigation. When this system has been successfully set up, we will compare it with other systems in the future in terms of performance.

References

Anderson, T. E., M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang (1996) Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1), 41-79.

Banino, J. S., J. C. Fabre, M. Guillemont, G. Morisset, and M. Rozier (1985) Some fault-tolerant aspects of the Chorus distributed system. *Proceedings of the 5th IEEE International Conference on Distributed Computing Systems*, pp. 430-437. Denver, CO, U.S.A.

Bloch, J. J., D. S. Daniels, and A. Z. Spector (1987) A weighted voting algorithm for replicated directories. *Journal of ACM*, **34**(4), 859-909.

Chen, P. M., E. K. Lee, G. A. Ginson, R. H. Katz, and D. A. Patterson (1994) RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2), 145-185.

Cheng, H. C. and J. P. Sheu (1991) Design and implementation of a distributed file system. *Software-Practice and Experience*, **21**(7), 657-675.

Dasgupta, P., R. J. LeBlanc, Jr., and W. F. Appelbe (1988) The Clouds distributed operating system: functional description, implementation details and related work. *8th IEEE International Conference on Distributed Computing Systems*, pp. 2-9. San Jose, CA, U.S.A.

Devarakonda, M., B. Kish, and A. Mohindra (1996) Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, **14**(3), 287-310.

Gifford, D. K. (1979) Weighted voting for replicated data. *7th ACM Symp. on Operating System Principles*, pp. 150-159. Pacific Grove, CA, U.S.A.

Guerraoui, R. and A. Schiper (1997) Software-based replication for fault tolerance. *IEEE Computer*, **30**(4), 68-74.

Howard, J. H., M. L. 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), 51-81.

Kistler, J. J. and M. Satyanarayanan (1992) Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, **10**(l), 3-25.

Purdin, T. D. M., R. D. Schlichting, and G. R. Andrews (1987) A file replication facility for Berkeley unix. *Software-Practice and Experience*, **17**(12), 923-940.

Satyanarayanan, M. (1990) Scalable, secure, and highly available distributed file access. *IEEE Computers*, **23**(5), 9-21.

Walker, B., G. Popek, R. English, C. Kline, and G. Thiel (1983) The Locus distributed operating system. *9th ACM Symp. on Operating System Principles*, pp. 49-70. Bretton Woods, NH, U.S.A.

Welch, B. and J. Ousterhout (1986) Prefix tables: a simple mechanism for locating files in a distributed system. *6th IEEE International Conference on Distributed Computing Systems*, pp. 184-189. Cambridge, MA, U.S.A.

設計一分散式檔案系統資料複製之容錯模式

陳宗禧* 張志勇** 許健平*** 游國忠***

*長榮管理學院資訊管理學系

**淡水工商管理學院資訊科學系

***國立中央大學資訊工程學系

摘 要

在這篇文章中，我們提出一個在分散式檔案系統中爲資料複製的容錯模式。該容錯模式結合了模組重覆（modular redundancy）、主從（primary-stand-by）及權重優先（weighted priority）三種技巧的優點。爲了滿足容錯的需求， 我們採用以目錄爲導向的複製及兩階層Heap之擴充前置表格（extended prefix table）的資料結構建構我們的容錯模式。在實際製作上，藉由Heap動態容易調整的特性，我們可以非常方便的操作該樹狀結構達到容錯之能力。我們所設計的容錯模式提供非常多的特性與能力，包括具較低的系統負荷、具多樣穿透性的能力、具容錯能力、具較佳系統執行效率等。