Highly Concurrent Directory Management in the GALAXY
Distributed System

Xiaohua Jia, Hirohiko Nakano, Kentaro Shimizu and Mamoru Maekawa
Department of Information Science
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

ABSTRACT

This paper presents a new efficient way of consistency control of replicated directories. By taking the advantages of special characteristics of directories, it achieves fast access to directories and high concurrency of updating directory replicas. The algorithm differs from conventional mechanisms for concurrency control of replicated data in two aspects: (1) It does not use global locks nor global timestamp orderings. (2) Updating operations can proceed without synchronizing to each other. The algorithm can survive from both node failure and network failure. The correctness and applicability of the algorithm are also discussed.

1. Introduction

In distributed systems, information objects such as files and active objects such as processes are distributed among many computing nodes. Directories are the means to locate these objects. Because accesses to objects can be requested from many nodes, directories need to be replicated for faster access and high reliability. This replication can be further complicated by the replication of file themselves. It is one of the important goals of distributed systems to design a fully distributed directory management with fast access, high concurrency and fault tolerance.

Many approaches for concurrency control, such as global locking, transaction processing, and timestamp ordering, have been developed to maintain the consistency of replicated data objects, but they are not particularly suitable for directory management in a large distributed system because the degree of replication of directories is very high and directories are distributed onto a large number of nodes.

In this paper, we propose an approach for concurrency control in directory management. This approach takes advantage of special characteristics of directories. A directory file consists of structured entries. Each entry is a collection of small items, and updating it can be done incrementally or differentially; that is, an updating can be done by a series of insertion and deletion operations.

Our approach has the following characteristics:
(1) Neither global locking nor timestamp ordering is required.
(2) Updating operations can be issued and executed with no need of synchronizing each other.
(3) Accesses to directories are allowed even during an insertion of a new object replica or a directory replica as well as during a deletion of an object replica or a directory replica.

The rest of the paper is organized as follows. The directory problem, design objectives and related works are described in Section 2. In Section 3, we define the system model and consistency control requirements. The data structures and algorithm are presented in Section 4. Section 5 discusses the fault tolerance and recovery mechanism of our approach. The applicability of the algorithm is discussed in Section 6. In Section 7, we evaluate our approach and compare with other works. The detail algorithm and consistency proof are given in Appendices 1 and 2.

2. Directory Management in Distributed Systems

2.1. Directory Problem

In an information management system, a directory structure is used to map a name into another piece of information such as the location of the data corresponding to the name. For example, a file directory maps a file name into the location of the file. In a centralized system, each entry of a directory is a pair of a name and the location of the object entity corresponding to the name. In a distributed system, objects may be replicated at different nodes, so the location information in a directory entry is a list of pointers to the object replicas. Since directories are heavily accessed by many nodes, the directory entries themselves may also need to be replicated. A directory entry should also maintain the information about the directory replicas. We assume each directory entry has three fields:
(1) Object name, the index to the entry of directories.
(2) Object location information, a list of node numbers that indicate the locations of object replicas, denoted as DIR (name).objs.
(3) Directory location information, a list of node numbers that indicate the locations of directory entry replicas, denoted as \( \text{DIR (name).dirs} \).

![Figure 1. Directory Structure](image)

Figure 1 shows the structure of a replicated directory entry. This structure is general and applicable to most of directory structures. In this paper, the directory file means an object identifier table \( (ID\ Table) \). Each directory entry is an \( ID \) entry. The name used to index an entry is a system-wide unique \( ID \).

The location information in a directory entry should be updated as replicas are created, deleted or migrated. The directory problem is how to maintain the consistency of the replicas of directory entries. The problem is made complicated by the fact that replicas of directory entries may be dynamically created and deleted at different nodes.

A directory entry is initially created in the system when a new object is created. After that, each creation of a replica of an object or an entry causes inserting a new member in the object or directory location list of the entry and each deletion of a replica causes deleting a member from the location list. When all the replicas of an object is deleted from the system, the directory entry is reclaimed by the system.

### 2.2. Objectives of Directory Manager

A directory is used as an index to access an object. It is referenced very frequently. The efficiency of directory accesses directly affects the total data access time. From this critical requirement, we list the objectives of directory management of a large distributed system as follows:

1. Achieving high concurrency.
2. Minimizing the waiting time for a process to access a directory.
3. Minimizing the number of message passing required for updating.
4. Allowing directory replicas themselves to be dynamically created and deleted.
5. Tolerating the node and network failures.
6. Being independent of the low level communication protocols and other facilities, such as globally synchronized clocks.

7. The consistency control policy of files can be designed independently of the consistency control policy of the directories.

### 2.3. Related Works

Bernstein [1,2] used two-phase lock mechanisms for modifications of directories and the insertions and deletions of directory replicas. Global locking has several obvious drawbacks, such as poor concurrency, high overhead and the burden for deadlock detection and avoidance.

Timestamp ordering is well used in concurrency control of replicated data in distributed systems [3,9], since it achieves better concurrency than the lock mechanisms in short transactions [7]. But it still does not improve the concurrency very much because of its need to serialize the commit of the transactions. Another drawback is the low efficiency resulting from its aborting the conflict transactions that already began their executions. The last is the difficulty of synchronizing clocks in a large distributed system.

Sunil Sarin describes a flexible algorithm for directory management [14]. A weighted voting mechanism is used for reading and updating replicated directories. The read operation is time consuming since it globally votes for the most recent data. The write operation inherits the drawbacks of global locking because replicas are locked during the voting for write.

The drawbacks of global locking, timestamp ordering and voting mechanism will become more serious in large distributed systems. They are not applicable to a large system.

The transaction processing for concurrency control is not suitable for this directory problem, because transaction processing requires synchronizing all the related nodes before committing the transaction. The directories of objects are accessed very frequently. If each access requires synchronizing with other nodes, the efficiency should be very low. Especially in a large distributed system in which nodes are connected by a WAN, this kind of synchronization is not realistic.

C.A.Ellis [4] and T.J.Wuu [5] raise a similar problem for directory management. High concurrency and fast response to updating are important requirements in their design. Each node invokes updating operation independently without synchronizing with others. In [5], to achieve consistency of replicated data in an unreliable network, Wuu proposed a method that maintains a log to record the requests that a node has served for each other node. C.A.Ellis proposed a concurrent edit system in [4]. In order to ensure that all the replicas have the same text while allowing each replica to be edited independently, it uses a transformation matrix to transform the conflicting operations from remote to the local operations. But both of them assume that the number of replicas in the system is fixed.
3. Model of Directory Manager

3.1. Definition

The system consists of \( n \) nodes, \( N_1, N_2, \ldots, N_n \), in the network. Each node can communicate with each other. \( \text{DIR}(\text{name})_N \) denotes the directory replica of \text{name} residing at \( N_i \). \( \text{OBJ}(\text{name})_N \) denotes the object replica of \text{name} residing at \( N_i \).

Directory manager resides in each node as a system server. It serves both local users and remote directory managers.

Directory manager provides the following operations.

1. \( \text{read(name,f,buffer)} \) returns the buffer with the contents of \( \text{DIR}(\text{name})_N.f \), where \( f \) is a field in the directory structure, \( f \in \{ \text{dirs}, \text{objs} \} \).
2. \( \text{insert(name,f,newnode)} \), \( \text{DIR}(\text{name})_N.f \rightarrow \{ \text{newnode} \} \). A new node can only be issued by the node who creates a replica of the directory or the object at the \text{newnode}. Here we call the original replica the parent and the created replica a child.
3. \( \text{delete(name,f,oldnode)} \), \( \text{DIR}(\text{name})_N.f \rightarrow \{ \text{oldnode} \} \).

In Section 6, we discuss the correctness of the \text{read} operation. Here we only discuss the consistency control concerning the two updating operations.

The migration of an object replica can be implemented by creating a new replica at the destination node and by deleting the old replica at the source node. In the remaining discussion, we assume the following:

1. An \( \text{insert(name,f,newnode)} \) operation can only be issued by the node who creates a replica of the directory or the object at the \text{newnode}. Here we call the original replica the parent and the created replica a child.
2. A \( \text{delete(name,f,oldnode)} \) can be invoked only by the \text{oldnode}. That means only the node from which a directory or an object is deleted has the authority to issue a \text{delete} operation.

These assumptions do not lose generality of our approach.

3.2. Serialization Control

From the above discussion, update operations defined on a directory entry are two incremental operations: \( \text{insert(name,f,mem)} \) and \( \text{delete(name,f,mem)} \). When they are performed on the same member \text{mem} of a field \( f \), they should be serialized in exactly the same order as they are invoked.

Two operations \( op_1 \) and \( op_2 \), if their executions are required to be serialized, are called serialized operations, denoted by \( op_1 \rightarrow op_2 \) as shown in Table 1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>\text{insert(name,f,mem)}</th>
<th>\text{delete(name,f,mem)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{insert(name,f,mem)}</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>\text{delete(name,f,mem)}</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1. Serialization Requirements

The "No" in the table means the case does not occur. Since the directory manager of a node does not allow any other node to make a replica at its node while the same replica still exists locally.

Serialization Control: Two operations \( op_1 \) and \( op_2 \), if \( op_1 \rightarrow op_2 \) holds, then at any node in \( \text{DIR}(\text{name})_N \), \( op_1 \) should be executed prior to \( op_2 \).

4. Data Structures and Algorithm

4.1. Necessary Information and Data structures

Each node, say \( N_i \), maintains the following information and data structures for directory operations:

- **Local Clock**
  - \( N_i \) maintains a local clock denoted by \( C_{N_i}(t) \) the value of \( C_{N_i} \) at physical time \( t \). Local clocks need not be synchronized to each other in the system.

- **Directory Structure**
  - A directory entry of object \text{name} in node \( N_i \) contains two lists, \( \text{DIR}(\text{name})_N.dirs \) and \( \text{DIR}(\text{name})_N.objs \).
  - \( \text{DIR}(\text{name})_N.dirs \) is a list of \( (N_i,N_p,C_{N_i}(t)) \) indicating that a directory replica of the object resides at \( N_i \), \( N_p \) is the creator of the replica and \( C_{N_i}(t) \) is the time when the node \( N_i \) learns the existence.
  - \( \text{DIR}(\text{name})_N.objs \) is a list of pairs \( (N_i,N_p) \) indicating that replica \( \text{OBJ}(\text{name})_N \) is created by \( N_p \).

- **Data Structures for Directory Manager**
  - \( \text{Req-rec}[n] \) is an array containing the requests sent to \( N_i \) but have not been acknowledged by \( N_i \). Each entry in \( \text{Req-rec}[N_i] \) is a pair \( (op,C_{N_i}(t)) \), consisting of an operation and a local timestamp indicating when the operation is issued. The elements are sorted by their timestamps.
  - \( \text{Reg-rec}[n] \) is an array containing the requests sent to \( N_i \) but have not been acknowledged by \( N_i \). Each entry in \( \text{Reg-rec}[N_i] \) is a pair \( (op,C_{N_i}(t)) \), consisting of an operation and a local timestamp indicating when the operation is issued. The elements are sorted by their timestamps.

4.2. Basic Approach

We first define the following:

Definition1. A completion state: For any node \( N_i \), \( i = 1, 2, \ldots, n \), all operations requested by \( N_i \) have been acknowledged and \( N_i \) does not issue any new updating operation at this time.

Definition2. Consistency: Directories in the system are said to be consistent if all the replicas of a directory entry have the same contents when the system is in a completion state.
To guarantee the consistency of all the replicas of a particular directory, we design the approach defined by the following four rules:

A 1. The request operations coming from a node are executed in the same order as they are issued at that node.

A 2. An updating operation op (name) requested by a node will be executed exactly once at all the nodes in DIR (name).dirs.

A 3. If two operations op1 and op2 are op1 \rightarrow op2, then at any node, op1 should be executed prior to op2.

A 4.1. Node Ni receives insert (name, dirs, Ni) from Np:
- build a descendant set of Ni in the directory entry:
  \[ \text{Child}_{Ni} = \{ N_1 | (N_1, N_i, C_{Ni}(t)) \in \text{DIR}(\text{name}).\text{dirs} \}; \]
- initially, Descend = \{ Ni \};
- for any \( N_i \in \text{Descend}, \text{Descend} = \text{Child}_{Ni} \cup \text{Descend} \).
- for any \( N_q \in \text{Descend}, N_i \text{ sends to } N_q \text{ the op (name) that} \text{arrives at } N_p \text{ after } N_p \text{ has created a copy at } N_i \text{ and that have not yet been sent to } N_q \).

A 4.2. Node Ni receives delete (name, dirs, Ni) from Np:
- \( N_i \) makes Ni's children as the children of its parentNi's:
  \( \text{suppose that } (N_1, N_i, C_{Ni}(t)) \in \text{DIR}(\text{name}).\text{dirs} \),
  \( \text{for any } N_j, \text{if } (N_j, N_i, C_{Ni}(t)) \in \text{DIR}(\text{name}).\text{dirs}, \text{make it into } (N_j, N_p, C_{Ni}(t)) \).

A short proof that A 1, A 2, A 3 and A 4 guarantee the consistency is given in Appendix 2.

4.3. Algorithm

The algorithm is designed to implement the above rules A 1 – A 4 defined in Subsection 4.2.

Parameters

In the following description, the following parameters are used:
- \( N_i \): local node;
- \( N_j \): remote node;
- \( N_p \): the parent of \( \text{DIR}(\text{na})_N \) or \( \text{OBJ}(\text{na})_N \);
- \( t_{\text{now}} \): current physical time;
- \( f \): name of a field in a directory entry, \( f \in \{ \text{dirs}, \text{objs} \} \);
- \( \text{na} \): name of an object, an index to the directory entry.

Rules A 1 and A 2

For implementing A 1, we take the sending-history policy to send requests, i.e. each time we send a request, we send all the requests that have not been acknowledged. For example, if a locally generated request is going to be sent to \( N_i \), the request is first put into \( \text{Req_rec}[N_i] \), then all the requests in \( \text{Req_rec}[N_i] \) are sent to \( N_i \) together.

Local Server:
1. Get an operation request from local interface.
2. Send the request together with the request history to all replicas.

At the other side, the remote server of a directory manager is waiting for any arriving request message. Because of our sending-history policy, an operation may be contained in more than one arriving message. To avoid re-executing the same operation, i.e. for A 2, the service record is used to trace on the last request operation it has served for each node. In order to ensure that the messages arrive at remote node in the same order as they are sent out, any "acknowledge" is sent together with the request history, in the same way as a request.

Remote Server:
1. Receive a request message from a remote node.
2. Execute the operations that have not been executed before.
3. Record the timestamp of the last operation it served.
4. Acknowledge the request with that timestamp.

Rules A 3 and A 4

The procedures Do_Insert and Do_Delete in Appendix 1 are the implementations of A 4.1 and A 4.2 respectively. They are called by the remote server when a remote insert or delete request is executed. For ensuring the serialized operations to be executed in a correct order, i.e. for A 3, they first check whether the coming insert or delete request arrives too late or too early according to the serialization requirements. If it is too early, enqueue the request waiting for the arrival of its partner of serialization. If it is too late, dequeue the request. The enqueue and dequeue conditions are as follows:

- If a request insert (na, \( N_k \)) arrives, while \( N_k \) still exists in \( \text{DIR}(\text{na})_N \), then enqueue the insert (na, \( N_k \)) in \( \text{Mch}_{\text{que}}[N_k] \).
- If a request delete (na, \( N_k \)) arrives, but \( N_k \) is not in \( \text{DIR}(\text{na})_N \), suppose that \( N_p \) is the parent of \( N_k \)’s \( \text{DIR}(\text{na}) \) or \( \text{OBJ}(\text{na}) \), then enqueue the delete (na, \( N_k \)) in \( \text{Mch}_{\text{que}}[N_p] \).
- If a request insert (na, \( N_k \)), comes from \( N_j \), and delete (na, \( N_k \)) \in \( \text{Mch}_{\text{que}}[N_j] \) then, the first delete (na, \( N_k \)) from the head of \( \text{Mch}_{\text{que}}[N_j] \) exactly matches with the coming insert (na, \( N_k \)); the same thing happens if a delete (na, \( N_k \)) comes from \( N_k \) and insert (na, \( N_k \)) \in \( \text{Mch}_{\text{que}}[N_k] \).

In dealing with the insertion or deletion of a directory replica, a special care should be taken. Do_Insert sends to the newly inserted \( \text{DIR}(\text{na}) \) and its descendants the operations they have missed; Do_Delete makes the children of the deleted \( \text{DIR}(\text{na}) \) as the children of its parent's and from the \( \text{Req_rec} \) delete the requests still being tried to be sent to the deleted replica.

The details of the algorithm are in Appendix 1.

5. Fault Detection and Recovery

In the approach described above, each node holds the requests to the other nodes until it makes sure that the requests have been properly executed. This mechanism makes the fault recovery very easy. Since every modification is invoked by the request message in the system, if a node is recovering from a failure, so long as it receives all the request messages that it had missed during
its failure, it would recover to the consistent state with others.

Both node failure and network failure are discussed here. To make the information of Req_rec at each node survive from failure, a copy of Req_rec is always kept in a permanent storage. We assume that the node failure does not destroy the information in the permanent storage.

For a node failure, we consider two cases, one is a proper shutdown, and the other is an accidental crash. In both cases, the system will invoke a recovery procedure when it is recovering. In order not to affect other nodes’ recoveries, before a node commits a shutdown, it moves its Req_rec to the leader of its group or a safe node. But the system suffered from the accidental crash cannot do this operation. We assume that the physical recovery of any accidental crash will be conducted soon. The recovery procedure of a node, say \( N_i \), requests all the other nodes for the missing request messages, and the node which receives this request sends its local Req_rec \([N_i]\) as a reply.

To detect a network failure and to recover the failure as soon as possible, each node periodically sends test messages to the nodes from which it has not received a shutdown message but cannot communicate with. As soon as the communication becomes available, the testing node sends the saved requests to the nodes which have suffered from a network fault to bring them to the most recent state.

6. Applicability

The directories may be in transient states during the system’s running, and the replicas of a directory entry may not be consistent. We will show that in spite of this, by using the information obtained from the local directory, the system can properly locate any object entity or replicas of the object.

First we see how to locate an object in the case of object migration. Assume an object \( \text{Obj}(id) \) with identifier \( id \) has been migrated from \( N_1 \) to \( N_2, \ldots, N_f \), \( N_f \neq N_i, 1 \leq i \leq f \), and a process \( P \) at \( N_i \) wants to access \( \text{Obj}(id) \), but the local \( \text{DIR}(id) \) is still \( N_1 \). That is; \( N_i \) does not get the operation of migrating \( \text{Obj}(id) \) out of \( N_1 \). According to our algorithm, \( \text{insert}(id, \text{obj}, N_2) \), \( \text{delete}(id, \text{obj}, N_1) \) \subset Req_rec \([N_i]\) in \( N_2 \). When a \( N_f \)’s request arrives at \( N_1, N_f \) finds that \( \text{Obj}(id) \) is migrated to \( N_f \), then \( N_f \), going on in this way, \( N_f \) will finally find \( \text{Obj}(id) \) at \( N_f \).

A file system uses the directory of a file object to find the total number of replicas and their locations in the system, so as to maintain the consistency of file replicas. The consistency control on replicated files can be independent of the scheme used on directory replicas. Here we only show that consistency control mechanisms on replicated files can work properly with the file replica information obtained from the directory.

For generality, assume that the weighted voting mechanism is used for consistency control on replicated files and the vote assigned to a replica at its creation does not change until the replica is deleted. Each file replica is required to carry the following information essential for voting: vote version, vote, read quorum and write quorum, denoted by \( v, r, w \), respectively. The vote version is only valid for the voting information, different from the version that indicates the recency of the content of the file replica.

Usually the voting mechanism requires that all replicas of an object be known in advance and a creation or deletion of replica is not allowed while a read or write voting is being conducted, because it will alter the read or write quorum of the object. This degrades the access performance. Here we give a simple solution, which allows read or write operations to proceed concurrently with insertions and deletions of directory replicas or object replicas.

The file system is required to obtain at least \( \max(r,w) \) votes for its committing an update on the voting information of an object, so that any nodes which have not learned the change of the quorum can find the new quorum before they obtain \( r \) or \( w \) since \( r \) (or \( w \)) + \( \max(r,w) \) > total votes of the object.

Suppose the file server at \( N_p \) creates a replica at \( N_k \) (or deletes a local replica). It takes the following steps:

1. read the voting information from the local file replica, initializing: \( \text{Sum} = v; \text{recent}_v = \text{vers}; \text{recent}_r = r; \text{recent}_w = w \).

2. request the directory manager \( \text{read}(\text{na.obj.obj-list}) \) for file replica list \( \text{obj-list} \).

3. broadcast a voting request to \( \text{obj-list} \), waiting for replies.

At the other side when a remote server receives the request, it replies a message containing the queued requests for \( N_p \) in its directory manager and the voting information of the local replica if they are there.

4. receive a reply, processing it as follows:
   - If there are any requests of updating directories in the reply, ask directory manager to process them and if any \( \text{insert}(\text{na.obj}.N_i) \) in the requests, send to \( N_i \) the same voting request as in step 3.
   - If the voting information is in the reply, meaning that a replica still exists there, retrieve the \( v, \text{vers}, r, w \) and adjust variables:
     \[
     \text{Sum} = \text{Sum} + v;
     \text{recent}_v = \text{vers}; \text{recent}_r = r; \text{recent}_w = w.
     \]

5. if \( \text{Sum} \geq \max(v, r, w) \) then repeat step 4, otherwise recalculate the read and write quorum \( \text{new}_r \) and \( \text{new}_w \) as taking the vote of the new replica into count, and \( \text{new}_v = \text{new}_v + 1 \); write \( \text{new}_v, \text{new}_r \) and \( \text{new}_w \) to the replicas who replied.

6. request the directory manager for \( \text{insert}(\text{na.obj}.N_k) \).

Because any existing replica can be visited finally and the number of replicas is finite in the system, step 4 will finish in a finite time. A deletion takes similar steps. We will not describe it here.
Now we see how the file system maintains the consistency of replicas. When the file system is going to write or read a file, the same as voting for an insertion or deletion of a file replica, it reads the file replica list from the local directory and sends voting request for write or read to the replicas. During the time it collects votes, if it finds that any new replica is created, it sends the voting request to the new replica. Since an insertion or deletion of file replicas requires the quorum of $\max(r,w)$, any node which obtains the quorum of $r$ or $w$ should find a more recent quorum. So finally if it obtains the votes greater than $w$ (or $r$) with the most recent version it has obtained so far, this obtained most recent version should be the most recent version of the quorums of the object in the system, it can commit the write (read).

7. Evaluation and Conclusion

7.1. Evaluation

We evaluate our algorithm mainly from the three aspects: concurrency, efficiency and fault tolerance.

In our algorithm, updating operations can be issued concurrently with no need of synchronizing to each other; and their executions can be proceeded to the end without being blocked or aborted by others. Centralized mechanisms, such as token passing, primary copy, do not allow concurrently issuing update operations; and distributed mechanisms such as locking, timestamp ordering, weighted voting, allow at most one of the conflicting update operations to proceed to the end.

In distributed systems, data access efficiency has two criteria, access time and the number of network message passing. First we see the blocking time of a process to read and update a replica. In our approach, read operations are always performed on the local replica, thus almost no delay is incurred. An update operation is issued in a non-synchronous way and sent to other nodes with no need to reach a consensus in advance. The control is returned to the user who requests the update immediately. Comparing with reaching a network-wide consensus, the blocking time of users is much less. While in the mechanisms involving network-wide synchronizations, the blocking time of an update is the maximum round-trip time from local to any other node in the system. Voting mechanisms may improve the performance for update, but it blocks user's read request for a network-wide consensus.

Then we see the number of message passings for read and update operations. Our approach does not take network communication for read operations; and for each update operation, it sends to other replicas a request message containing all the operations that have not been acknowledged and waits for the replies from all the other replicas, so each updating requires $2(m-1)$ message passings, where $m$ is the number of replicas in the system. A locking mechanism requires $3^*(m-1)$ messages for each update. A voting mechanism at least requires $(m-1)\ast 2^w$ messages for an update and $(m-1)\ast r$ for a read request.

Fault tolerance is another important requirement of distributed algorithms. Our approach works as long as there is at least one node running properly in the system. It can recover the nodes from both node and network failures with fairly less message passing in the network. A global locking mechanism cannot work if one node is in failure. A voting mechanism requires that at least $\max(r,w)$ nodes are in running and can communicate to each other. Available copies[11] can survive as long as one node is running, but it cannot work at the presence of network partition. Virtual network partition[8] can tolerate both node and network failures, but it at most allows the nodes in one partition to access the replicas of data.

Comparing with other consistency control mechanisms on data replication, our approach achieves better performance in the management of replicated directories.

7.2. Conclusion

This paper has presented a new efficient concurrency control method for replicated directories. It has three special features: no use of global locking and global timestamp ordering, fast access, and fault tolerance. Those features are very important for large distributed systems. In a large system, any global lock or any kind of global timestamps are unrealistic because of large communication delays and a huge number of nodes.

The GALAXY system, which is under development in our laboratory, is a large distributed operating system. The proposed directory management system is implemented in GALAXY and works properly.

References

Appendix 1. Algorithm

Local Server:
get an op (na) from local interface
execute the op (na) on local DIR (na)
for each Nk ∈ DIR (na).dirs begin
  Req_rec [Nk] := (op (na),CNk(tm))
  send (Nk, Req_rec [Nk])
endfor

Remote Server:
receive a request message from a remote node Np
for (op,CNk(tm)) ∈ message
New_ops = {(op,CNk(tm)) | CNk(tm) > Sv_rec [Np]}
for each (op,CNk(tm)) ∈ New_ops:
  if op == insert/delete then
call Do_Insert/Do_Delete
else if op == "ack (CNk(tm))" then
  Req_rec [Np] := (op,CNk(tm))
  CNk(tm) = max{CNk(tm) ∈ New_ops}
  Sv_rec [Np] = CNk(tm)
  Req_rec [Np] += { (ack(CNk(tm)), CNk(tm)) }
  send(Np,Req_rec [Np])

Do_Insert(NA f Nk);
/* the request is from Np */
if "delete (NA f Nk)" ∈ Mch_que [Np] then
dequeue delete and return
if f == dirs then
  Descend = {Nk and Nk’s descendants
  in DIR (NA).dirs }
  for each Nk ∈ Descend:
    send Np the requests that arrives Np after Np
    creates the DIR (NA)Np and have not been sent to
    Np.
if Nk ∈ DIR (NA).f then
  enqueue insert in Mch_que [Nk]
if f == dirs then
  DIR (NA).dirs += (Nk, Np, CNk(tm))
else DIR (NA).objs += (Nk, Np)

Do_Delete (NA f Nk);
/* Np is parent of Nk in DIR (NA).f */
if insert (NA f Nk) ∈ Mch_que [Nk] then
dequeue insert and return
if NOT (Nk ∈ DIR (NA).f) then
  enqueue delete in Mch_que [Np]
if f == dirs then
  Req_rec [Np] := (op (NA),CNk(tm))
  make Nk’s children as the children of its parent Np ‘s
  DIR (NA).f := (Nk, Np)
Appendix 2. Correctness Proof

The following assertion will be used without proof.

Assertion: Two sets $S_1$ and $S_2$, initially $S_1 = S_2$. We define:

Two operations performed on the sets:

- **insert(S,mem)**
  - if NOT(mem) $\in S$ then $S += mem$ else ERROR
  - delete(S,mem)

For two given operation sequences $Ops_1 = \{op_{11}, \ldots, op_{1n}\}$ and $Ops_2 = \{op_{21}, \ldots, op_{2m}\}$, where $op_{ij} \in \{insert,delete\}$.

$Ops_1 = Ops_2$, but operations may be in a different order in $Ops_1$ and $Ops_2$. The serialization requirement of $insert(S,mem) - delete(S,mem)$ hold both in $Ops_1$ and $Ops_2$. After performing $Ops_1$ on $S_1$ and $Ops_2$ on $S_2$, the result should be:

$$S_1 = S_2$$

**Lemma: A 1.6 & A 2.6 & A 3.6 & A 4 – Consistency Definition**

Proof. Let $op(na_{\lambda(N_k,t)})$ denote the operations on $DIR(na)$ issued by $N_k$ and arriving $N_t$ after $t$.

For a given $na$, assume all $DIR(na)$s are initially consistent, and there is a virtual node in the system. $DIR(\lambda)_{vir}$ is the $DIR(na)$ at the virtual node, $DIR(na)_{\lambda\omega,n}$ will never be deleted and can receive any operation on $DIR(na)$ in the system. $DIR(na)_{\lambda\omega,n}$ is initialized to be the parent of the current exiting $DIR(na)$s in the system.

After the system runs for a time, it reaches a completion state.

![Figure 2. Parent-Children Relation](image)

Suppose during running, $DIR(na_{\lambda})$ is created by $N_p$ at time $t$; See Figure 2. $DIR(na_{\lambda}, dirs) = \{N_1, N_2, \ldots, N_m, N_p\}$ at time $t$. A 2 ensures: $insert(na_{dirs}, N_i)$ issued by $N_p$ will be last executed at $N_1, N_2, \ldots, N_m$.

For any $N_k \in \{N_1, N_2, \ldots, N_m\}$, suppose the $insert(N_i)$ arrives at $N_k$ at $t_k$. We first consider that $DIR(na_{\lambda})$ is already known by $N_k$. By A 4.1, when $N_k$ receives the $insert(na_{dirs}, N_i)$, $N_k$ sends to $N_i$ $op(na_{\lambda\omega,n})$ and $N_i$ knows $N_k$'s existence from $t_k$.

Then we inspect $N_k$'s new children. Suppose the newly created children are $N_{k_1}, N_{k_2}, N_{k_l}$ and $\{N_{k_1}, N_{k_2}, N_{k_l}\}$.

![Image 423](image)