オペレーティング・システム 48－9
（１９９０．　９．　７）

Ｇａｌａｘｙ分散ＯＳにおける並列ディレクトリ管理

賈　小華　　中野　裕彦　　清水　謙多郎　　前川　守

東京大学理学部　情報科学科

本論文では、ＧＡＬＡＸＹ分散オペレーティング・システムのネットワーク透過な名前付けと位置付けの機構について述べる。ＧＡＬＡＸＹでは、ＩＤ　Ｔａｂｌｅと呼ばれる大域的なデータベースを介して、その部分的なコピーを各ネットワーク・ノードにもたせることにより、オブジェクトの存在する位置を直接求めることが可能である。本論文ではとくに各ノードがもつＩＤ　Ｔａｂｌｅのコピー（レプリカ）の効率的かつ信頼性の高い一貫性制御について述べる。ＩＤ　Ｔａｂｌｅの性質を利用して、並列性の高い効率的な制御を行なう。またこの機構はノードおよびネットワークの障害に対応することができる

ID Table Mechanism for Object Naming and Locating in the GALAXY Distributed Operating 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 discusses a network transparent naming and direct object locating mechanism by means of ID table in GALAXY system. It also presents a new efficient way of consistency control of replicated ID table entries, By taking the advantages of special characteristics of ID table, the algorithm of ID management achieves high degree of concurrency and efficiency of accessing IDTE replicas. The algorithm can survive from both node failures and network failures.

ID Table Mechanism for Object Naming and Locating in the GALAXY Distributed Operating 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 discusses a network transparent naming and direct object locating mechanism by means of ID table in GALAXY system. It also presents a new efficient way of consistency control of replicated ID table entries, By taking the advantages of special characteristics of ID table, the algorithm of ID management achieves high degree of concurrency and efficiency of accessing IDTE replicas. The algorithm can survive from both node failures and network failures.

1. Introduction

Naming plays an important role in the design of any operating system. Especially in case of distributed operating systems, where a large number of objects are distributed all over the system, the need to assign global names to all the objects makes naming a more important issue. Object locating is an important function of naming mechanisms. Because the naming mechanism supports references to objects, it directly influences both the ease with which users refer to objects, and efficiency of locating objects.

Object identifiers (ID for short) are commonly used to uniquely identify objects in distributed systems. ID Table is a structure consisting of many ID table entries (IDTE for short). Each IDTE contains the location information of each object in the system. Because accesses to objects can be requested from many nodes, ID table entries need to be replicated for faster access and higher reliability. This replication can be further complicated by the replication of file themselves. It is important to design a fully distributed management of ID table with fast access, high concurrency and fault tolerancy.

This paper discusses the mechanism for realizing network-transparent object naming and locating in the GALAXY distributed operating system; We also propose an approach for concurrency and consistency control of ID table management. This approach takes advantage of special characteristics of ID table, and thus achieves high degree of concurrency and efficiency of ID table access.

2. Object Manager

In GALAXY, there are following basic types of objects: processes, files, storages, devices, nodes and users. Any type of basic objects is managed by a special module dedicated to the type. We call such a module by a general term *object manager*. For example, process objects are managed by a Process Manager, file objects are managed by a File Manager, and so on.

Object Managers reside on all the nodes where the objects of that type exist; each manages a subset of the objects on the node, and they all cooperatively manage all the objects of the type. When an operation invocation message is issued to an object, the corresponding object manager is invoked.

Each object is assigned a system-wide unique ID. In order to guarantee the uniqueness of each ID, the ID format consists of two fields: *time stamp (TS)* field and *node number (NN)* field. Each field consists of 8 bytes. The TS field contains the time stamp assigned to the ID by the node that has created the ID. The NN field contains the node number where the ID is created. For better performance, we define an eight-byte (two-word) short-format ID which is only effective in each local node. The short-format ID is a real-time time stamp and it has a *remote bit* to indicate whether the object is in local node.

3. Three-Level Naming Scheme

Names are used to designate or refer to objects at all levels of the system architecture. They have various purposes, forms and properties depending on the levels at which they are defined. However an informal distinction can be made between two basic classes of names widely used in operating systems: *human-oriented names* and *system-oriented names*.

Human-oriented names are required to meet the needs of human users for their own mnemonic names, and to assist them in organizing, relating, and sharing objects. Therefore human-oriented names should be flexible enough to allow a user to define his/her own names rather than simply identify the object, and they should be independent of the physical location and structure of objects which they designate. System-oriented names are automatically generated by the system and they are used either by the users or by the system. In many cases, system-oriented names are uniquely defined and have the uniform format for the management purpose. Both the human-oriented name and the system-oriented name are resolved within a global and distributed context.

The GALAXY system is designed based on the above naming model. In GALAXY, human-oriented names are called *external names*. The basic structure of external names is hierarchical but attributes can be attached to it for representing the semantic relations among objects. An external name is completely independent of the physical location and structure of objects. It is translated by the user-defined naming contexts. The naming contexts may be global, and there may be multiple naming contexts in the system. The intentions to allow multiple naming contexts are flexibility and efficiency[11].

As for system-oriented names, in GALAXY, every object has an unique ID. The context of IDs is global and unique in the system. An ID identifies an object, but its structure and management mechanism are irrelevant to the contents, external name, or physical addresses of the object's replicas. In GALAXY, one object can have multiple replicas. All replicas of an object use the same ID irrespective of their locations.

4. ID Based Object Locating

The IDs of all the objects in the system are stored in a system-wide table called *ID Table*. The ID Table contains all the information necessary for accessing the corresponding objects. It consists of many IDTEs. Each IDTE contains the a list of node numbers that indicate the locations of object replicas. It is not efficient and reliable to let some *central* node keeps the entire ID Table. Conversely, it is also not realistic for every node to have a copy of the entire ID Table. Thus, in GALAXY, each node has a partial copy of the ID Table. The copy of the ID Table of a particular node contains entries for the following IDs:

(1) IDs that are contained in the naming contexts (directory files) on the node. This is necessary because when an external name is resolved to an ID, that ID

must be available in the node otherwise further operations for locating that object cannot be carried out.

(2) IDs that are used by the processes running on the node. The necessity of these IDs at this node is obvious.

Availability of the entries for these two categories of IDs in the copy of a particular node's ID Table ensures the direct locating of any object from any node.

At each node, a separate *local object table* is maintained for each type of object that resides on that node. The actual physical location of an object within a node is maintained in the local object table. Each entry of an object table consists of:

(1) short-format ID

(2) physical address of an object

(3) access control information

The format of a physical address varies according to the type of an object. For example, it contains the device number(s) and block number(s) for file objects.

Locating objects is divided into two steps:

1. to get the number of the node where the object exists
2. to get the physical address where the object exists

Step (1) is realized by the ID Table which contains the number of the node where the object exits. Note that due to the specific replication criteria of IDTEs at different nodes, this method does not require any network communication and the object locating process is performed locally for all objects. In step (2), each object manager performs the locating operation in a local node by using the object table[11].

5. Model of ID Table Management

5.1. Definition

In GALAXY system, IDTE is replicated for faster object locating. Furthermore, object may be replicated for better access efficiency and reliability. Each IDTE has three fields:
(1) Unique ID.
(2) Object replication list, a list of node numbers that indicate the locations of object replicas, denoted as $IDTE(id).objs$.
(3) IDTE replication list, a list of node numbers that indicate the locations of IDTE replicas, denoted as $IDTE(id).ids$.

The location information in an IDTE should be updated as replicas are created, deleted or migrated. The ID manager is to maintain the consistency of the replicas of IDTEs when one of the replica is updated.

Assume that the system consists of n nodes, $N_1, N_2, , , N_n$, in the network. Each node can communicate with each other. $IDTE(id)_{N_i}$ denotes the IDTE replica of id residing at N_i. $OBJ(id)_{N_i}$ denotes the object replica of id residing at N_i.

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

ID manager provides the following operations:

(1) $read(id.f, buffer)$, it returns the buffer with the contents of $IDTE(id).f$, where f is a field in the IDTE structure, $f \in \{ids, objs\}$.

(2) $insert(id.f, newnode)$,
$IDTE(id).f = IDTE(id).f + \{newnode\}$.

(3) $delete(id.f, oldnode)$,
$IDTE(id).f = IDTE(id).f - \{oldnode\}$.

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 *insert*(*id*,*f*,*newnode*) operation can only be issued by the node who creates a replica of the IDTE or the object at the *newnode*; Here we call the original replica the parent and the created replica a child.

(2) A *delete*(*id*,*f*,*oldnode*) can be invoked only by the *oldnode*. That means only the node from which a IDTE or an object is deleted has the authority to issue a *delete* operation.

These assumptions do not lose generality of our approach.

5.2. Serialization Control

From the above discussion, update operations defined on an IDTE are two incremental operations: *insert*(*id*,*f*,*mem*) and *delete*(*id*,*f*,*mem*).

When they are performed on the same member *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.

	insert(id,f,mem)	*delete(id,f,mem)*
insert(id,f,mem)	No	→
delete(id,f,mem)	→	No

Table 1. Serialization Requirements

The "No" in the table means the case does not occur. Since the ID 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 *IDTE*(*id*).*ids*, op_1 should be executed prior to op_2.

6. Data Structures and Algorithm of ID Management

6.1. Necessary Information and Data structures

Each node, say N_l maintains the following information and data structures for IDTE operations:

Local Clock

N_l maintains a local clock denoted by C_{N_l}. $C_{N_l}(t)$ is the value of C_{N_l} at physical time t. Local clocks need not be synchronized to each other in the system.

IDTE structure

An IDTE of object *id* in node N_l contains two lists, *IDTE*(*id*).*ids* and *IDTE*(*id*).*objs*. *IDTE*(*id*).*ids* is a list of 3_tuples $(N_i, N_p, C_{N_l}(t))$, indicating that a IDTE replica of the object resides at N_i, N_p is the creator of the replica and $C_{N_l}(t)$ is the time when the node N_l learns its existence. *IDTE*(*id*).*objs* is a list of pairs (N_i, N_p), indicating that replica $OBJ(id)_{N_i}$ is created by N_p.

Data Structures for ID Manager

(1) Request record

Request record is an array *Req_rec*[*n*] of size *n*. An entry *Req_rec*[N_i], corresponding to node N_i, contains the requests sent to N_i but have not been acknowledged by N_i. Each element in *Req_rec*[N_i] is a pair $(op, C_{N_l}(t))$, consisting of an operation and a local timestamp indicating when the operation is issued. The elements are sorted by their timestamps.

(2) Service record

Service record is an array *Sv_rec*[*n*] of size *n*, an entry *Sv_rec*[N_i], corresponding to node N_i, is a timestamp C_{N_i} of the last request operation that the node N_l has served for N_i.

(3) Match queue

Match queue is a queue $Mch\_que[n]$ of size n, an entry $Mch\_que[N_i]$, corresponding to node N_i, contains the serialized operations that arrive at N_l too early and wait for the arrivals of matching operations of their partners from N_i.

6.2. Basic Approach

We first define the following:

Definition1. A completion state: For any node N_i, i=1, 2, , , 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: IDTEs in the system are said to be consistent if all the replicas of an IDTE have the same contents when the system is in a completion state.

To guarantee the consistency of all the replicas of a particular IDTE, we design the approach defined by the following four rules:

$A1$ The request operations coming from a node are executed in the same order as they are issued at that node.

$A2$ An updating operation $op(id)$ requested by a node will be executed exactly once at all the nodes in $IDTE(id).ids$.

$A3$ If two operations op_1 and op_2 are $op_1 \rightarrow op_2$, then at any node, op_1 should be executed prior to op_2.

$A4.1$ Node N_l receives $insert(id, ids, N_i)$ from N_p:

- build a descendant set of N_i in the IDTE:
  $Child = \{N_c \mid (N_c, N_i, C_{N_l}(t)) \in IDTE(id).ids\}$;
  initially, $Descend = \{N_i\}$;
  for any $N_j \in Descend$, $Descend = Child \cup Descend$.
- for any $N_d \in Descend$, N_l sends to N_d the $op(id)$s that arrive at N_p after N_p has created a copy at N_i and that have not yet been sent to N_d.

$A4.2$ Node N_l receives $delete(id, ids, N_i)$ from N_i:

- N_l makes N_i's children as the children of its parent N_p's:
  suppose $(N_i, N_p, C_{N_l}(t)) \in IDTE(id).ids$,
  for any N_j that
  $(N_j, N_i, C_{N_l}(t)) \in IDTE(id).ids$:
  make it into $(N_j, N_p, C_{N_l}(t))$.

A proof that $A1$,$A2$,$A3$ and $A4$ guarantee the consistency is given in [7].

6.3. Algorithm

The algorithm is designed to implement the above rules $A1 \sim A4$ defined in Subsection 6.2.

Parameters

In the following description, the following parameters are used:

N_l: local node;

N_r: remote node;

N_p: the parent of $IDTE(id)_{N_i}$ or $OBJ(id)_{N_i}$;

t_{now}: current physical time;

f : name of a field in an IDTE, $f \in \{ids, objs\}$;

id: unique ID of an object, an index to the IDTE.

Rules $A1$ and $A2$

For implementing $A1$, 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_r, the request is first put into $Req\_rec[N_r]$, then all the requests in $Req\_rec[N_r]$ are sent to N_r 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 an ID 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 $A2$, 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 $A3$ and $A4$

Two procedures Do_Insert and Do_Delete are the implementations of $A4.1$ and $A4.2$ 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 $A3$, 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, f, N_k) arrives, while N_k still exists in *IDTE*$(id).f$, then enqueue the *insert*(id, f, N_k) in *Mch_que*$[N_k]$.

If a request *delete*(id, f, N_k) arrives, but N_k is not in *IDTE*$(id).f$, suppose that N_p is the parent of N_k's *IDTE*(id) or *OBJ*(id), then enqueue the *delete*(id, f, N_k) in *Mch_que*$[N_p]$.

If a request *insert*(id, f, N_k), comes from N_i, and *delete*$(id, f, N_k) \in$ *Mch_que*$[N_i]$ then, the first *delete*(id, f, N_k) from the head of *Mch_que*$[N_i]$ exactly matches with the coming *insert*(id, f, N_k); the same thing happens if a *delete*(id, f, N_k) comes from N_k and *insert*$(id, f, N_k) \in$ *Mch_que*$[N_k]$.

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

6.4. 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.

7. Applicability of ID Management

The IDTEs may be in transient states during the system's running, and the replicas of an IDTE may not be consistent. We will show that in spite of this, by using the information obtained from the local IDTE, 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 $Obj(id)$ with identifier id has been migrated from N_1 to N_2, , , to N_f, ($N_i \neq N_j$, $1 \leq i,j \leq f$), and a process P at N_k wants to access $Obj(id)$, but the local $IDTE(id).objs$ is still N_1. That is; N_k does not get the operation of migrating $Obj(id)$ out of N_1. According to our algorithm, $\{insert(id,obj,N_2), delete(id,obj,N_1)\} \subseteq Req\_rec[N_k]$ in N_1. When a N_k's request arrives at N_1, N_k finds that $Obj(id)$ is migrated to N_2, then N_k goes to N_2, going on in this way, N_k will finally find $Obj(id)$ at N_f.

A file system can use the local IDTE information 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 IDTE replicas. For generality, the weighted voting mechanism is assumed to be used for consistency control on replicated files. An algorithm using the local IDTE locating information to maintain the consistency of replicated files is given in [7].

8. Evaluation and Conclusion

The three-level naming mechanism in GALAXY system provides a network transparent naming facility to users. The multiple naming context in the mechanism makes it achieving better flexibility, scalability and usability. The locating mechanism allows to locate any object in the system right at an accessing node without inquiring from remote nodes, which makes the locating mechanism very fast and also improves the overall system performance by reducing the network traffic.

The ID manager has the following advantages: high degree of concurrency and efficiency, and fault tolerancy. In the ID management, 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. Distributed mechanisms such as two-phase locking[1,2], timestamp ordering[3,6], weighted voting[4,9,10], allow at most one of the conflicting update operations to proceed to the end.

The efficiency of access IDTE is high. In our approach, read operations are always performed on the local replica and 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 blocking time of a read and update operation is much less than those with network-wide synchronizations, such as global locking, timestamp ordering, voting, and so on. The network-wide

massage passing of updates in out algorithm is also low.

Fault tolerancy is another advantage of our ID management. 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[9,10] mechanism requires that at least $max(r,w)$ nodes are in running and can communicate to each other. Available copies[8] can survive as long as one node is running, but it cannot work at the presence of network partition. Virtual network partition[5] 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 algorithm of ID management achieves better performance in managing replicated IDTEs.

References

1. Philip A.Bernstein, Vassos Hadzilacos and Nathan Goodman, Concurrency Control and Recovery in Database System. P289-307, Addison_Wesley. 1987.
2. Philip A.Bernstein and Nathan Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. ACM Trans. on Database System, Vol.9,No.4, Dec.1984, P596-615.
3 J.B.Rothnie, P.A.Bernstein, N.Goodman,etc. System for Distributed Database (SDD-1). ACM Trans. on Database System, Vol.5,No.1,Mar. 1980, P1-17.
4. D.K.Gifford. Weighted Voting for Replicated Data. Proc. Seventh ACM Symposium on Operating System Principles, 1979.
5. A.E.Abbadi, D.Skeen and F.Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. Proc. of the 4th ACM symp. on PODC.1985, P215-229.
6. A.Demers, D.Greene, etc. Epidemic Algorithms for Replicated Database Maintenance. Proc. of the 6th ACM Symp. on PODC, 1987. P1-12.
7. X.Jia, H.Nakano, K.Shimizu and M.Maekawa. Highly Concurrent Directory Management in the GALAXY Distributed System. ICDCS, 1990.
8. J.L.Carroll, D.D.E.Long, J.Paris. Block-Level Consistency of Replicated Files. Proc. of ICDCS. 1988. P146-153.
9. D.Barbara, H.Garcia-Molina and A.Spauster. Policies for Dynamic Vote Reassignment. Proc. of ICDCS. May 1986. P37-43.
10 S.Sarin, R.Floyd, and N.Phadnis. A Flexible Algorithm for Replicated Directory Management. Proc. of ICDCS. 1989. P456-464.
11 P.K.Sinha, K.Shimizu, N.Utsunomiya, H.Nakano and M.Maekawa, Network-Transparant Object Naming and Locating in GALAXY Distributed Operating System. Submitted to Journal of Information Processing, Japan.