Regular Paper

Network-Transparent Object Naming and Locating in the GALAXY Distributed Operating System

PRADEEP K. SINHA*, KENTARO SHIMIZU*, NAOKI UTSUNOMIYA*, HIROHIKO NAKANO* and MAMORU MAEKAWA*

This paper describes the concepts and mechanisms for realizing network-transparent object naming and locating in distributed systems. Multiple global naming contexts and descriptive naming based on hierarchical names are important features for increasing the flexibility of naming for human users. The proposed mechanism for locating objects by means of their system-wide unique identifiers, which are used by many existing distributed operating systems, makes it possible to locate any object in the system right at an accessing node. It can be applied to general global flat name spaces. This paper analyzes the efficiency of this locating mechanism and its variations in comparison with conventional object locating mechanisms. This paper also describes the implementation issues of object naming and locating in the GALAXY distributed operating system now being developed by the authors.

1. Introduction

Naming plays an important role in the design of any operating system. Especially in the case of distributed operating systems, where a large number of objects are distributed throughout the system, the need to assign global names to all the objects makes naming an important issue. Locating objects 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 the efficiency of object locating.

This paper discusses the concepts and mechanisms for realizing network-transparent object naming and locating in distributed operating systems. Multiple global naming contexts and descriptive naming based on hierarchical names are used in our approach. These are desirable features of names in distributed operating systems, because they lead to better efficiency, flexibility, and usability. We also discuss a mechanism for locating objects by means of their system-wide unique identifiers, which are used in many existing distributed operating systems. Unlike conventional locating mechanisms, this mechanism allows any object in the system to be located exactly at an accessing node, without the need to inquire from remote nodes where the object exists. This makes the object locating mechanism very fast and also improves the overall system performance by reducing the network traffic. Our naming and locating mechanisms aim at transparency with respect to the physical location and structure of objects. This is one of the most important requirements in distributed operating systems. The concepts and mechanisms proposed in this paper are developed in the GALAXY [9, 12, 24] distributed operating system that we are now implementing, and are applicable to other distributed/non-distributed systems.

2. GALAXY's Objects

There are several ways of considering objects. In the pure smalltalk-like view, objects represent physical entities, such as boxes and cars. Thus in pure object-oriented systems, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft or a car. An object consists of some private memory that holds its state. The private memory is made up of the values of a collection of instance variables. The value of an instance variable is itself an object and therefore has its own private memory for its state (namely, its instance variables). A primitive object, such as an integer or a string, has no instance variables. It only has a value, which itself is an object. More complex objects contain instance variables, which in turn contain other instance variables.

GALAXY is not a pure object-oriented system, and hence the granularity of object identity is not at integer or string level. In GALAXY, only the entities that need to be identified at the operating system level, such as processes, files, devices, and nodes, are viewed as objects. Thus GALAXY's primitive objects are normally very large in comparison with the primitive objects of pure object-oriented systems. The main reason for the choice of relatively large primitive objects in GALAXY

*Department of Information Science, Faculty of Science, university of Tokyo.

is to facilitate the design and implementation of the operating system. Granular primitive objects may be used in the design of the data base systems or the languages supported by the system on top of the operating system level. This paper deals only with the design aspects of the GALAXY distributed operating system. Thus the naming and the locating mechanisms discussed in this paper are mainly designed for objects to be dealt with at the opeating system level.

The objects in our system basically have the following two attributes:

1. *Object Identity*: Objects have an existence independent of their contents or values. Thus, two objects can be either identical, in which case they are the same object, or they can be equal, in which case they have the same contents or values. Identical objects are known as replicas in GALAXY.

2. *Object Type*: Every object in GALAXY belongs to a particular type. A type describes a set of objects with the same characteristics. It describes the structure of data carried by objects as well as the operations (*methods* in object-oriented terminology) applied to these objects. Users of a type see only the interface of the type, that is, a list of methods together with their signatures (the type of input parameters and the type of the result): this is called *encapsulation*.

In GALAXY, each type of object is managed by a special module dedicated to the type. We refer to such a module by the 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 at which the objects of that type exist; each one cooperatively manages a subset of the objects on the node. When an operation invocation message is issued to an object, the corresponding object manager is invoked.

3. The Simple Naming Model

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 an object, and they should be independent of the physical location and structure of objects they designate. The facilities of alias, context-based naming, and grouping are widely used for human-oriented names. System-oriented names are automatically generated by the system and are used either by the users or by the system. In many

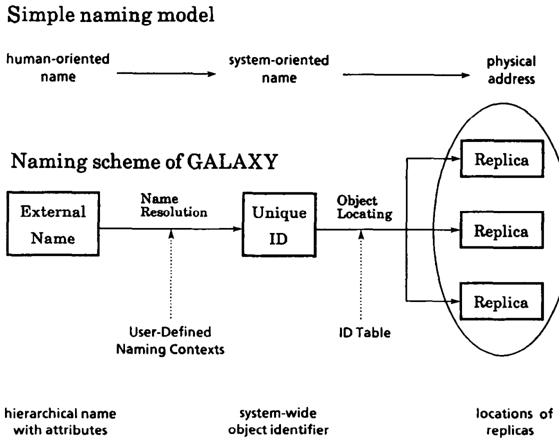


Fig. 1 Simple naming model and the naming scheme of GALAXY.

cases, system-oriented names are uniquely defined and have a uniform format for management purposes. Both human-oriented names and system-oriented names are resolved within a global and distributed context, but they have distinct naming and locating mechanisms in many distributed operating systems [4, 7, 10, 11, 22].

Figure 1 shows a simple naming model for distributed operating systems. In this naming model, a human-oriented name is translated into a system-oriented name, and a system-oriented name is translated into the physical addresses of replicas. The GALAXY distributed operating system is designed according to the above naming model. The details of our naming and locating mechanisms are described in the following sections.

4. Human-Oriented Object Naming

In GALAXY, human-oriented names are called *external names*. Network transparency is one of the most important requirements of the naming scheme in distributed operating systems. In addition, efficiency and flexibility of use are equally important. Basically, there are two approaches to global naming in conventional distributed operating systems:

(1) Using a separate name space for each node

(2) Using a single global name space for all nodes.

Early network operating systems such as Newcastle Connection [3] and COCANET [20] used the first approach to global naming. They did not have the property of location transparency, because the location of an object had to be explicitly incorporated in its name.

Many recent distributed file systems, such as NFS [21] and RFS [18], use the first approach along with a method called *remote mounting* to achieve the goal of location transparency in naming. In these systems, each node in the network has its own root file system in which remote file systems are locally mounted. The problem with these systems is that the same object may have different *absolute* names when viewed from different nodes. This lack of an absolute naming facility can cause difficulties for distributed application programs, since processes running on different nodes are in different naming domains. For example, an application using several nodes to process a data file would run into trouble if the file name were specified as */user1/project1/data1*; rather than opening the same file, participating processes on nodes having identifiers *rome* and *paris* would respectively open */rome/user1/project1/data1* and */paris/user1/project1/data1*. Even if the user were to explicitly specify */paris/user1/project1/data1*, the program would fail if *rome* did not have *paris's* file system mounted, or worse, had some other node's file system mounted under the name */paris*. Thus, although systems using remote mounting support transparency with respect to the location of the *accessing object* (such as a process), they do not support transparency with respect to the location of the *accessed object*. This transparency problem of such systems may be solved by mounting all the file systems of all the nodes on the same position in the name hierarchy of each node. However, in this case, if there are n nodes, then n^2 mounts have to be performed. This makes the management very difficult, especially for very large networks. Thus, this method does not work well for systems having a large number of nodes.

Many recent distributed operating systems such as DOMAIN [11], LOCUS [28], ANDREW [13], PULSE [10], ELXSI [15] and Saguaro [1] use the second approach to the transparent global naming of objects. In these systems, requests from all the nodes are served by first searching a global name space. Thus an object's absolute name is always the same, irrespective of the node from which it is used for accessing the object. Although many of these systems support transparency with respect to both the location of the accessing object and the location of the accessed object, the main problem with them is the inadequacy of their name resolution mechanisms. Their name resolution mechanisms suffer from one or more of the following drawbacks:

(1) The overhead involved in name resolution is high because directory objects are cached with page as a unit.

(2) The flat structure of the name caches affects the search efficiency of large caches.

(3) Creation of name caches on a per-process basis may cause duplication of cache creation overhead and duplicate cache entries on the same node.

(4) The scalability of the mechanism is poor, which limits its applicability to small networks.

To handle the transparency feature, we use the single global name space approach in GALAXY. However, to overcome the problems inherent in other systems that use the single global name space, GALAXY supports an efficient and scalable name resolution mechanism, details of which are given in Section 6. In addition to this, GALAXY also provides facilities for multiple global naming contexts and descriptive naming. The reasons for this are to increase flexibility and efficiency, as discussed below.

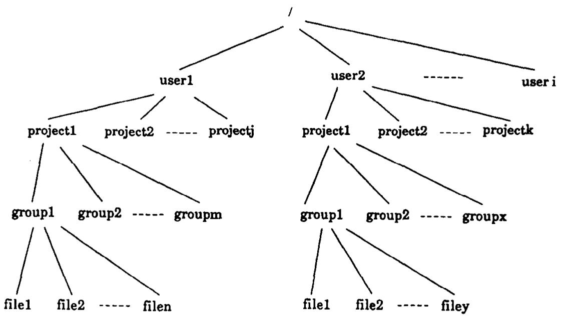


Fig. 2 A typical name tree.

4.1 Multiple Global Naming Contexts

To avoid the problems of inconvenience of use and inefficient resolution of long pathnames in a single global name space, the first step taken in GALAXY is to provide users with flexibility in defining their own naming contexts. A *context* is basically a pathname of a single global name tree, starting from its root. For example, for the name tree shown in Fig. 2, the pathname */user1/project1/group1* represents a typical context. Similarly, the pathnames /, /user1, /user2/project2, /user1/project1/group2, and /user2/project1/group1/file1, also represent other contexts for the same name tree.

A GALAXY user can define his/her own contextname for a particular pathname. For example, a particular user may specify that the pathname */user1/project1/group1* be designated as *mycontext1*. Now, when that user wants to use the object with the pathname */user1/project1/group1/file1*, instead of specifying the complete pathname, he/she can simply specify the contextname *mycontext1* and the remaining components of the object's pathname, *file1* in this case. To facilitate this, the basic naming syntax of an external name in GALAXY is

[Contextname]Pathname

where contextname is the name of one of the global contexts of the system and pathname is a list of component names separated by '/'. For our example above, once the contextname *mycontext1* has been defined, instead of using the full pathname */user1/project1/group1/file1* to specify the external name of the desired object, the external name for the same object may simply be specified as *[mycontext1]file1*. Depending upon his/her needs, a particular user may specify several such contextnames for his/her use.

Thus, GALAXY's external names may be of two types: *absolute* and *relative*. In its absolute form, an external name consists of the complete pathname of the corresponding object starting from its root. On the other hand, in its relative form, an external name consists of a contextname and a pathname that is relative to the given context. Both forms are acceptable and may be used to specify object names.

4.1.1 Management of Contextnames

For the management of user-defined contextnames, we use the concept of *node groups*. In this method, the entire system is hierarchically partitioned into small groups of nodes. Each node group has a group leader, which is responsible for maintaining the information about the nodes belonging to its group. In addition to other information, the leader node also maintains information about all the contextnames defined at the nodes belonging to this group. That is, the leader node maintains a *group context table*, which is a mapping of all the contextnames and the corresponding nodes belonging to its group. Thus, when a user defines a new contextname, the information about that contextname is stored in the local context table of the node at which the contextname is defined and also in the group context table of the leader node of the group to which this node belongs. At the leader node, the mapping table contains only the contextname and the corresponding node number to which the contextname belongs. But the mapping table of the node to which the contextname belongs contains the contextname and the system-wide identifier of the directory file corresponding to the contextname.

Now when a user (say at node A2 in Figure 3) specifies a contextname to be used, the search for that contextname proceeds as follows:

Step 1: The context table at the local node (node A2) is searched.

Step 2: If the contextname is not found in Step 1, the group context table at the leader node of the group (node A) is searched.

Step 3: If the contextname is not found in Step 2, the group context table at the leader node of the parent group (node D) is searched.

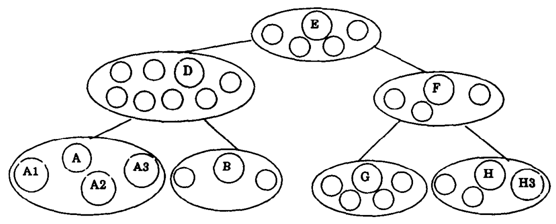
Step 4: If the contextname is not found in Step 3, the group context tables at the leader nodes of the groups that are first cousins of the group whose leader is A (node B is the only first cousin in Fig. 3) are searched one by one.

Thus, the searching process continues like this in a hierarchical manner until the location of the desired contextname is found. For example, in Fig. 3, if a user at node A2 wants to use a contextname defined at a node, say H3, which belongs to the group whose leader is H, then the searching sequence will be

A2->A->D->B->E->F->G->H->H3

and a message will be returned from node H3 to node A2.

Note that the efficiency of this scheme lies in the proper grouping of the nodes. For example, nodes that very frequently interact and use each others' contextnames are placed in the same group. Groups that frequently interact are placed close to each other by making them close relatives. On the other hand, groups that never or rarely interact with each other are made distant relatives.



(a) An example of hierarchical node groups.

Contexts	Node Nos.
C1, C2, C3	A1
C4	A2
C5, C6	A3
C7, C8	A

(b) A group context table maintained by the leader node A of the node group comprising nodes A, A1, A2, and A3. Similar tables are maintained by all the leaders of all the node groups.

Contexts	Corresponding IDs
C1	ID for C1
C2	ID for C2
C3	ID for C3

(c) A local context table maintained at node A1, consisting of contexts defined locally and their corresponding IDs. Similar tables are maintained by each node for the locally defined contexts.

Fig. 3 Management of contexts by the method of node groups.

4.1.2 Problem of Duplicate Contextnames

Since the users of GALAXY are given full freedom to assign their own contextnames to the various contexts of the single global name space, there is a possibility that two or more users may assign the same contextname to different contexts of the name space. For example, in the name space of Fig. 2, user1 may assign a name *mycontext* to the context */user1/project1/group1* and user2 may assign the same name *mycontext* to a different context */user2/project1/group1*. With this type of contextname assignment, an external name *[mycontext]file1* may now refer to both of the objects */user1/project1/group1/file1* and */user2/project1/group1/file1*. The problem is to decide which object is actually being referred to by the user.

To solve this problem, that is, to identify contexts uniquely, a contextname is used along with the user_id (login name) of the user who defines it. Thus our previous basic syntax for specifying an external name changes to the form

[user_id: Contextname]Pathname

The default user_id is the user specifying the external name, and need not be specified along with the contextname when the user wants to refer to a context defined by himself/herself. With this addition, if user1 and user2 specify *[mycontext]file1*, this will now be interpreted by the system as */user1/project1/group1/file1* and */user2/project1/group1/file1*for user1 and user2 respectively. But if user1 wants to use the object */user2/project1/group1/file2*, then he/she must specify user2's identification name along with the contextname as *[user2:mycontext]file2*.

Note that the specification of user_id along with the contextname is necessary only when the same names are assigned to different contexts by different users. Obviously, the system does not allow the same user to define two contexts by the same name, and a particular user must assign unique names to the various contexts defined by himself/herself.

The system primitives in GALAXY that provide its users with full flexibility for context based naming are given in Fig. 4.

4.2 Descriptive Naming

Flat names with attributes are widely used in daily life. Personal names are a typical example of this type of naming. However, owing to the problem of uniqueness of name representation and difficulty in address mapping, most operating systems use hierarchical names instead of flat name spaces: UNIX [19], NFS [21], Sprite [14], LOCUS [28], V system [5] are a few examples of such systems. In hierarchical names, it is easy to represent the hierarchical structure of objects and to manage them as a group. However, hierarchical names cannot easily represent various relations among objects, and require a large overhead to resolve. In order to incorporate the advantages of hierarchical names and to overcome their disadvantages, GALAXY uses the hierarchical names as a basis and provides a facility for attaching naming attributes with links between two components of a hierarchical name. Naming attributes are the properties of the object being named. Such properties are represented as labels attached to the links between directories. By linking names, it is possible to define the semantic structure of objects. Such information is useful for some kinds of utility programs and application programs.

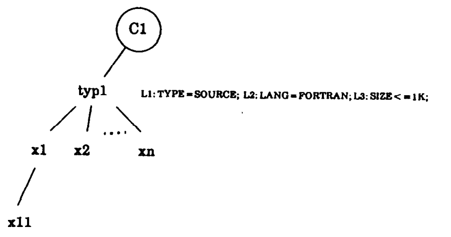
For example, in the name tree of Figure 5(a), the external name [C1]typ1 has the following attributes assigned to it:

L1: TYPE=SOURCE; L2: LANG=FORTRAN;
L3: SIZE<=1K;

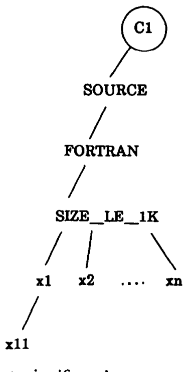
which signifies that all the files x1, x2, . . . , x*n*, x11, and so on that are descendants of the directory object corresponding to the specified external name have the abovementioned properties. Thus the picture of the type of object being referred to, say, by the name [C1]typ1/x1 will be clear to the user when he/she knows about the attributes listed above. If attribute-based naming is not used then this simple name does not signify anything to its users. Without the use of attribute-based naming, an attempt to assign meaningful names to objects will result in longer names. For example, Figure 5(b) shows

- CreateContext(*Contextname, ContextID*)
  Used to create a new context. The unique ID of the context is returned by the system.
- DeleteContext(*Contextname*)
  Used to delete the context specified by Contextname.
- LinkContext(*Contextname1, Pathname, Contextname2*)
  Used to link Contextname2 to Contextname1 under the directory specified by Pathname.
- Resolve(*Pathname, Contextname, ID*)
  Used to get the ID of Pathname in the context specified by Contextname.
- Bind(*Pathname, Contextname, ID*)
  Used to register Pathname in the context Contextname with ID.
  If Pathname includes the directories that do not exist, this primitive returns an error.
- Unbind(*Pathname, Contextname, ID*)
  Used to unbind Pathname in the context Contextname. The unique ID that was registered with the unbound name is returned.
- UnbindDirs(*Pathname, Contextname, ListofIDs*)
  Used to unbind all names under the directory specified by Pathname in the context Contextname. A list of unique IDs that were registered with the unbound names is returned.
- ChangeWorkingContext(*Contextname*)
  This primitive changes the current working context to the specified context.
- ChangeWorkingDirectory(*Pathname*)
  This primitive changes the current working directory to the specified Pathname.

Fig. 4 GALAXY's system primitives for context-based naming.



(a) Example of attribute assignment to an external name.



(b) Name tree that signifies the same attributes as Fig. 5(a) without using attribute-based naming.

Fig. 5 An example to illustrate the advantage of attribute-based naming.

the name tree that must be defined in order to signify the same attributes for objects x1, x2, . . . , x*n*, and so on as specified above. Note that in this case the pathname is SOURCE/FORTRAN/SIZE_LE_1K/x1 as compared to the previous pathname typ1/x1 for the same object x1 having the desired properties. Thus the descriptive naming facility allows users to define the semantic structure of objects without the necessity of using long object names. Short object names in turn contribute to efficient name resolution and are easy to use.

Xerox's Clearinghouse [16] is an example of an operating system that supports descriptive naming based on attribute name, attribute type, and attribute value. Its primary application is to store a profile of users. In Clearinghouse, the above 3-tuple is provided to allow attributes to be assigned to names. In addition, the hierarchical name structure consists of only three levels. GALAXY's approach is more general in the sense that the tree-structured names may have *n* levels of hierarchy and attributes can be attached to one or more levels. In this case, the object specified by a pathname possesses the attributes of all its component names.

The system primitives in GALAXY that provide its users with full flexibility for attribute-based naming are given in Fig. 6. Note that these primitives are similar to LISP functions.

· PutProperty(*Externalname, Label, Value*)
  Used to assign a label and the corresponding value to the specified external name

· GetProperty(*Externalname, Label, Value*)
  Used to get the value corresponding to the *Label* for the specified external name

· RemoveProperty(*Externalname, Label, Value*)
  Used to remove the property specified by *Label* for the specified external name

· GetProperties(*Externalname, ListofProperties*)
  This primitive returns a list of (label, value) pairs for the specified external name.

Fig. 6 GALAXY's system primitives for descriptive naming.

5. System-oriented Object Names

In GALAXY, every object has a unique identifier for system-oriented names called *Unique ID* (henceforth referred to as ID). An object's ID is unique in the entire system. In GALAXY, one object can have multiple replicas. All replicas of an object use the same ID, irrespective of their locations. 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.

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. In particular, an ID Table entry (called an IDTE) contains information on the type of object, the access control list for the object, the location of the object's replicas (*replica list*), and the locations in which copies of this IDTE exist (*copy list*). The replica list helps in returning all the locations of the desired object when the object is to be located for accessing. The copy list links all the IDTEs of the same object together so that any modification can be consistently made to all the copies through this link.

5.1 Generation of Unique IDs

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 eight 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 number of the node at which the ID is created.

The value of the TS field is created in the following way in order to guarantee the uniqueness of the TS, even if the node operates in a stand-alone manner and even if it crashes and is restarted:

1. The value of the hardware timer is set to the variable T.
2. The node's logical time, which is stored in a particular location on the disk, is set to the variable S.
3. The logical time S is obtained by the equation

$$S = \max\{T, S\} + R$$

where R must be much larger than the time required to store S on the disk.

It may be observed that even if the unit of time for incrementing the TS field is 1 μs, the time period represented by the TS field is $2^{64}\ \mu\text{s} \approx 3 \times 10^5$ years. Obviously, this time period is dominant over the life-time of the system. Similarly, the NN field is long enough to assign unique numbers to all the nodes.

6. Name Resolution

As shown in Fig. 1, and external name is first mapped to its corresponding ID, which in turn is mapped to the physical locations (node numbers) of the replicas of the object concerned. In this paper, we define *name resolution* as the process of mapping an external name to its corresponding ID, and *object locating* as the process of mapping an ID to the replica locations of the concerned object. In this section we will discuss the name resolution mechanism of GALAXY. The object-locating mechanism of GALAXY will be discussed in the next section.

6.1 Name Cache for Directory Entries

In conventional operating systems, directories are used to map an object's name to its physical location. Thus, in these systems, a directory entry consists of a component name and the corresponding object descriptor pointer, such as inodes in UNIX [19] and vnodes in NFS [21]. In GALAXY, on the other hand, a directory entry is a (*component name, ID*) pair, which maps the name of a component of an object to its system-wide unique ID. These directories are regular GALAXY objects that are distributed among the various nodes of the system and can be replicated and migrated just like any other object. In addition, *name caches* are used in GALAXY at each node, for caching necessary directory entries. At a particular node, a separate name cache is created for each context that corresponds to an object's name cached on that node. The name cache structure is the same as that of the hierarchical directory structure of the name space. A particular node's name cache consists of those directories and directory entries that correspond to the contextname and the component names of the pathname of an object that was recently (see the life of a name cache entry, described later in this section) used to access the object from the node. For example, as shown in Fig. 7, if an object with external name [C]a/b/c was recently accessed from the node, then the node will have a name cache for the context [C], which will have the following directories and directory entries corresponding to this object: (a) a directory for [C] having an entry for a; (b) a directory for [C]a having an entry for b; and (c) a directory for [C]a/b having an entry for c.

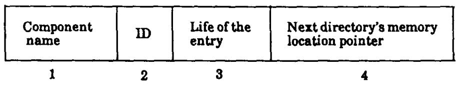
Although the name cache directories of a particular node appear to be similar to regular directories, they differ from them in the following aspects:

1. They normally contain very few entries in comparison with the corresponding regular directories, because only the entries required at a node are cached in the name cache of that node.
2. The name caches are always resident in the memory.

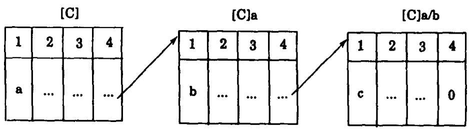
As shown in Fig. 7, in addition to the (component name, ID) pair, an entry of the name cache directory has the following fields:

1. *Life of the entry*: This is used to keep track of the access pattern of a name cache entry from that node. When a new entry is created in a name cache directory, the value of the field for that entry is initialized to a constant value L. Whenever a name cache entry is accessed, its life is reinitialized to L. That is, all cache entries are initially assigned a constant life and a particular cache entry gets a new life every time it is accessed. If the current time becomes greater than the life of a particular name cache entry, then that entry is called a *dead entry*.
3. *Next directory's memory location pointer*: If the component name of an entry of a name cache directory represents another directory object that is also cached in the same name cache, then the field points to the memory location of that directory in the same name cache. This field is used to trace the components of an object's pathname in the name cache.

New cache entries are created on a particular node at the time of locating an object one or more of whose pathname components or whose contextname is not cached on that node. The details of this are given in Section 6.2. A dead entry of a name cache is no more useful for the local node, and may be removed. Such entries are replaced by new entries while searching the space for new entries in the name cache. All the dead entries of a cached directory have the same possibility of being replaced, and hence the first dead entry encountered during the search operation is replaced by the new entry. If there are no dead entries to be replaced, new space is allocated for the new entry. Since only necessary entries are placed in the name caches of a particular node, we assume that the space cost for the name caches at a particular node is low enough to put a size limit on the name caches.



(a) The fields of a name cache entry.



(b) A name cache for context [C] containing entries for the pathname components of the object [C] a/b/c.

Fig. 7 The structure of a name cache.

6.2 The Name Resolution Mechanism

The basic mechanism used for name resolution in our approach is the method of remote pathname expansion [23]. However, the use of a name cache at each node helps to improve the efficiency of our name resolution mechanism. When an object is accessed from a particular node by using its external name, the pathname components of the external name are searched for in the local name cache corresponding to the context of the external name. The pathname components are searched for one by one in the name cache directories, just as in regular directories. For example, if a name [C] a/b/c is to be searched for, then the search is performed in the name cache for context [C]. In this name cache, the directory for '[C]' is first searched for the component name 'a', then the directory for '[C]a' is searched for the component name 'b', and finally the directory for '[C]a/b' is searched for the component name 'c'. As shown in Fig. 7, the next directory's memory location pointer field of a name cache entry provides the address of the next directory in the name cache to be searched.

If the pathname of the desired object consist of n component names of which n_1 were found in the local name cache, then for the remaining $(n-n_1)$ components, the searching has to be continued somewhere outside the local name cache. Therefore, the ID corresponding to the last component name found in the local name cache is extracted from the name cache and is searched for in the local ID Table to obtain the location of the corresponding object. This ID will certainly be available in the local ID Table due to our policy of replicating IDTEs at various nodes (discussed later in Section 7.2.1). A message is now sent to one of these nodes with the remaining $(n-n_1)$ pathname components for remote pathname expansion. The remaining pathname components are searched for in the relevant directories at the new node. The object locating operation continues in this way until all the pathname components have been resolved and the desired object's ID has been extracted. During the course of remote pathname expansion, the ID and the IDTE of the objects corresponding to the remaining pathname components that were not found in the client's name cache are accumulated in the message used for remote pathname expansion. This accumulated information is finally sent from the server node (the node on which the desired object is located) to the client node, where it is used to make new entries in the local name cache and the local ID Table for future use.

In case of a *complete miss*, when a name cache corresponding to the desired object's external name context is not found on the client node, the hierarchical node group method of Section 4.1.1 is used for searching the node (say N) in which the ID corresponding to the concerned context is stored. In this case, the name resolution process starts from node N instead of the client node, in a similar manner to that discussed above. As before, the information accumulated during the name resolution process is finally sent from the server node to the client node, where it is used for creating the necessary name cache and name cache entries and also for making necessary entries in the local ID Table for future use.

6.3 Name Cache Consistency

In our approach, the use of a (component name, ID) pair instead of a (component name, physical location) pair in a name cache entry reduces the number of cases in which a name cache entry becomes stale. For example, a name cache entry does not become stale when an object's location changes on account of the migration of that object. Thus, in our approach, a name cache entry may become stale only in the following two cases:

(a) When an object has been deleted from the system but its name mapping is still present in a name cache.

(b) When a component of a directory has been deleted or renamed, but the old component name is still present in a name cache.

In GALAXY, for updates of type (a), we use the method of on-use consistency control, and for updates of type (b) we devised a method called the gradual invalidation method.

For updates of type (a), stale cache entries need not be updated at the time of directory updation, because such entries will never cause a name to be mapped to the wrong object and can be detected at the time of use. The best approach is therefore to use on-use consistency of name caches for such types of updates. In this case, if a user tries to access the deleted object from a node on which the stale name cache entry is present, then our mechanism will extract the corresponding ID from the name cache. When the ID Table is searched for the obtained ID, this ID will not be found in the ID Table, indicating that the desired object does not exist and hence that the stale cache entries corresponding to this object must be deleted from the node's name cache. For on-use invalidation of such cache entries, there is no need for an object manager to maintain a list of names of the objects being managed by it.

For updates of type (b), the method of gradual invalidation is used for maintaining name cache consistency. The basic gradual invalidation method is similar to the broadcast method for strict consistency in the sense that when a directory is updated, a message is sent to all other nodes to invalidate their corresponding name cache entries. However, to avoid the cost of broadcasting in a large network, in this method, the invalidate message is not broadcasted to all the nodes at one time, but is gradually propagated to all the nodes.

GALAXY uses a modified form of the basic gradual invalidation method. Note that in GALAXY the invalidate message need not be sent to all the nodes of the system. This is because when a directory entry is updated, the IDTE corresponding to the ID of the updated directory entry can be used to determine the nodes on which that directory entry is possibly located. It was mentioned in Section 5 that the copy list of an IDTE consists of all the nodes on which the IDTE is replicated. Moreover, as discussed later in Section 7.2.1, an IDTE is replicated at a particular node only if either the corresponding ID is present in one of the directories on the node or in the name cache of that node or if that ID is being used by a process running on that node. Thus the list of nodes extracted from the copy list of the IDTE is always a superset of the set of nodes that contain the updated entry either in their name cache or in their replicated directories. In GALAXY, therefore, the update message for a directory update of type (b) is multicast only to the nodes present in the copy list entry of the corresponding IDTE. This greatly reduces the scope of multicasting. However, since an IDTE may be replicated on several nodes, instead of sending the update message to all these nodes at a time, the message is gradually propagated.

Because of the use of the gradual invalidation method, temporary inconsistency of the name cache entries exists during the time required to propagate the message from the node at which updating took place to the node at which the stale name cache entry is located. We will now show that this type of temporary inconsistency of the name cache entries does not have any serious adverse effect on the user jobs running in their own consistent environments.

The problems that may occur in our mechanism owing to the inconsistency of name cache entries fall into the following types:

(1) A name has already been deleted by a user work-

ing on a remote node, but the corresponding update has not yet been made in the local name cache. In this case, a local user may still access the object by using the name.

(2) A name has already been changed (the object has been renamed) by a user working on a remote node, but the corresponding update has not yet been made in the local name cache. In this case, a local user may still access the corresponding object by using its old name. We will discuss how to detect and cope with these problems.

In problem (a), if a name has been deleted because of the deletion of the object pointed to by that name, then the name cache inconsistency does not cause any problem for normal applications. This is because the use of the deleted name by the local user returns the ID of the deleted object, but when the system uses this ID to locate the object, he system realizes that the object has already been deleted. Thus, although there is some delay involved in informing the user that the desired object no longer exists, the name cache inconsistency does not cause the user any serious problem.

Another reason for problem (1) may be that a name has been deleted to prevent access to the object by that name. This is similar to problem (2) and may be handled in the same way as that problem.

In problem (2), it is quite logical to permit the local user to access the object by using its old name. This is because the local user's view (environment) does not change until he/she is informed about the change made. Thus, as long as a user on another node is unaware of the change made in the object's name, he/she will be allowed to access the object by its old name. However, when the name cache entry corresponding to the changed name is invalidated, the use of the old name by a user on that node will return an error to the user indicating that the old name is no longer valid. Thus the user will become aware of the change made and will know that from now on he/she cannot use this name to access the object.

Thus the temporary inconsistency of the name cache entries due to the use of the method of gradual invalidation does not cause any serious problem to the users, although in some cases it degrades the efficiency of detecting and informing the users about the correct status of object names within the system.

6.4 Name Resolution Efficiency

The efficiency of our name resolution mechanism is highly dependent on the degree to which locality is exhibited in the use of external names for locating objects from a particular node. In our mechanism, an external name is cached at a node in which it was recently used, and its cached pathname is used for subpath expansion in the near future for all other pathnames belonging to the same context. Thus if a significant level of locality exists in the use of pathnames of a particular context, then a high cache hit ratio will result, allowing local resolution of several names and a significantly long subpath expansion at the client node itself for many other object names.

Measurements made by Sheltzer [23] and Cheriton [5] clearly show that a high degree of locality exists in the use of pathnames for locating objects. In case of the V system, over about 24 days of 24-hour operation, an average cache hit ratio of 99.70% was recorded [5]. With the same amount of locality, our name resolution operation will be very efficient, because almost all the external names can be resolved at the client node. In fact, we expect an even better cache hit ratio than that of the V system, because in the latter, a prefix cache is created on a per-process basis, and a desired name is searched for only in the process's own cache. On the other hand, in GALAXY, centralized name caches are used for all the processes running on that node, and remain available for use by other processes even when the process that created them no longer exists. The use of centralized name caches may appear to be slightly inefficient from the point of view of the time required to search the cache. However, this problem will not normally occur in GALAXY, because of the use of separate name caches for each context. Even if the search efficiency is slightly degraded, the overall efficiency is improved by the increase in the cache hit ratio for all the processes of the node.

6.5 Name Resolution with Object as Unit

In many distributed systems, the unit used for name resolution is a group of objects rather than an individual object. For tree-structured names, subtrees are often used as the unit of name resolution. The mount tables in NFS [21] and RFS [18], and the prefix tables in Sprite [14, 30] are examples of such subtree-based name resolution mechanisms. In these systems, the name resolution can be performed with an object as a unit by specifying a single object name as a subtree name in the name resolution mechanism. However, this approach requires a significant overhead. One important advantage of group-based name resolution is efficiency, but it degrades network transparency, and the flexibility of object replication and migration. For example, in LOCUS [28], the unit of name resolution is a file system called *file group* and replicating a file to another node requires the establishment of a file group. The *zone* of DOMAIN [11] and the *domain* of ELXSI [15] are similar concepts and mechanisms used for name resolution.

It is obvious from the name resolution mechanism discussed above that in GALAXY, the unit of name resolution is taken as a single object instead of a group of objects. Thus any object can be replicted or migrated to any node in the system independently of any other object. That is, the replication or migration of a particular object does not require the concomitant establishment or mobility of any other object.

7. Object Locating

In this section we will discuss GALAXY's object locating mechanism, which deals with the process of mapping an object's ID to its replica locations.

7.1 Conventional Object Locating Mechanisms

The following types of object locating mechanism can be conceived in distributed operating systems.

(1) Broadcasting

As shown in Fig. 8(a), in the method of broadcasting, an access request for the desired object is broadcasted to all other nodes if it is not found in the local node. The node currently holding the desired object then replies to the accessing node. In this case, the amount of network traffic generated for each request is directly proportional to the number of nodes in the system and is prohibitive for large networks. However, because of its simplicity, the broadcasting method is used when the number of nodes is small, the communication speed is high, and the remote access request generated is not very frequent. Amoeba [26] uses this method for locating a remote port.

(2) Searching the creating node first and then broadcasting

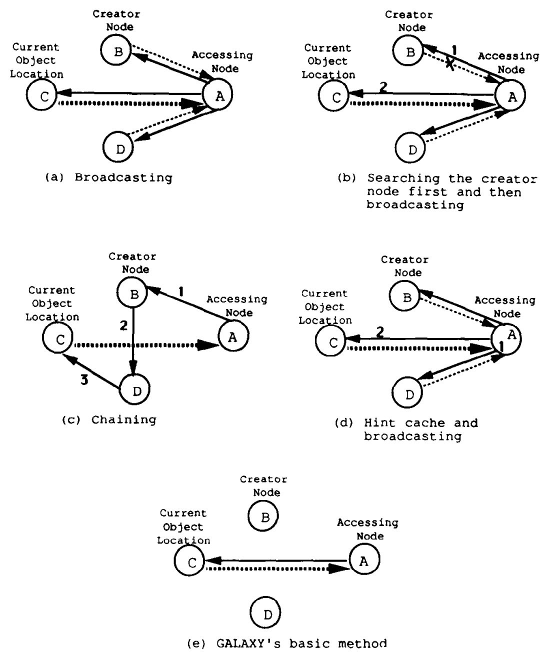
This method is based on the assumption that it is very likely that an object will remain at the node where it was created (although it may not be always true). Thus, as shown in Fig. 8(b), the node at which the desired object was created is first accessed, and in case of a failure, the request is broadcasted to other nodes. This method of locating an object is used by Cronus [22].

(3) Chaining

In this method, links are maintained to keep track of the present location of a particular object. When an object is migrated from one node to another, a pointer is maintained at its previous node to point to the object's new node. When an access request for a particular object is generated, the creating node is first accessed and the chain is then traversed until the current object location is reached. From the current object location, a reply is sent to the accessing node. The mechanism is illustrated in Fig. 8(c). The object locating cost in this case obviously depends on the frequency of object migration in the system. This method is used in DEMOS/MP [17], which was the first system to implement process migration. In practice, the method has the following two major disadvantages: (1) the object locating cost is directly proportional to the length of the link, and grows considerably as the link becomes longer, and (2) it is difficult, or even impossible, to access an object if an intermediate node in a link fails.

(4) Hint cache and broadcasting

In this method, each node maintains a cache that contains the present location of the object. Therefore, the accessing node first searches the cache in the local node for the desired object. If the object is not found at the local node, the method of broadcasting is used to locate the object. Figure 8(d) illustrates this mechanism. This method of object locating is used in V system [4, 5] and DOMAIN [11].


Creator Node
Current Object Location
Accessing Node
B
C
A
D
(a) Broadcasting
1
2
X
(b) Searching the creator node first and then broadcasting
3
(c) Chaining
(d) Hint cache and broadcasting
(e) GALAXY's basic method
Fig. 8 Object locating mechanisms.

7.2 Object Locating Mechanism in GALAXY

Unlike any of the above-mentioned approaches, GALAXY allows any object in the system to be located by using the information stored in the local node. Thus, as shown in Fig. 8(e), a direct accessing method is used in GALAXY for locating objects. In this method, the accessing node directly sends its request for the desired object to the current object location. This direct object locating mechanism of GALAXY is described in the following section.

7.2.1. Replication Policy for IDTEs

As discussed in Section 5, GALAXY's system-wide ID Table contains the IDs and the locating information of all the objects in GALAXY. In a usual non-distributed system, the ID Table can be managed in a centralized manner. But in a distributed multiple-host system such as GALAXY, it is not efficient and reliable for some *central* node to keep 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 directories on the node or in a name cache of that node. The presence of these IDTEs in the local ID Table of a node is necessary

Table 1 Management of reference counts in the IDTE.

Reference Count	Increment	Decrement
Directory entry (DRC)	Name definition Directory migration (in) Name cache entry creation	Name Deletion Directory migration (out) Name cache entry deletion
Accessing process (PRC)	Process creation Process migration (in)	Process deletion Process migration (out)

for the direct locating of the directory file objects during the name resolution process. This is because, when the pathname of an external name has been partially resolved at a node (say N), then the remaining pathname components are sent to the next directory location for which the IDTE corresponding to the next directory must be available at node N; otherwise the location of the next directory file object cannot be known at node N. To ensure the presence of these IDTEs in the node's ID Table, at the time of directory migration (in) or name cache entry creation at the node, the IDTEs corresponding to the concerned IDs are migrated into this node from the storage node of the concerned directory.

(2) IDs that are being used by the processes running on the node. These IDTEs are necessary in a node's ID Table for the direct locating of the objects being used by the processes of that node. Note that in GALAXY, when a process accesses an object for the first time, it uses the object's external name, which is resolved by the system, and the corresponding ID is returned to the process and is also entered into the ID Table of the process's node. Thus subsequent accesses to the same object by this process are made by using the object's ID. Therefore, in GALAXY, each process maintains a list of the IDs being used by the process, which is called the context of the process. When a process migrates from one node to another during the course of its execution, all the IDTEs corresponding to the objects belonging to the process's context are also migrated to the process's new node and entered into the ID Table of that node. This facilitates the direct locating of the object being used by a process, no matter how many times and to which node the process migrates during its course of execution.

The 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 when the object's ID is given. Direct locating of any object from any node is also possible if the entries for the IDs of all the objects are maintained in the copy of the ID Table of each node. However, this will lead to an enormous system overhead in terms of the space and consistency control of ID Table entries. To avoid such overheads, GALAXY ensures that each node maintains only the minimum required ID Table entries. For this purpose, each IDTE also has two kinds of reference count: *Directory Reference Count* (*DRC*) and *Process Reference Count* (*PRC*). The DRC indicates the number of local directories that contain the ID. The local name cache directories also contribute to the value of DRC. The PRC is the number of local processes that use the ID. Both DRC and PRC are managed locally. These two reference counts are managed as shown in Tale 1. An IDTE whose DRC and PRC are zero is regarded as unused. The unused IDTEs are actually deleted by a special daemon process, which is a part of the ID manager.

Since an IDTE is replicated at different nodes, it is necessary to maintain the consistency of all the replicas of an IDTE. In GALAXY, an incremental updating mechanism is used to maintain the consistency of all the replicas of an IDTE. The details of this mechanism have been presented in a separate paper [9]. In this paper, we assume that the consistency of IDTEs is maintained at all times.

7.2.2 The Basic Object Locating Mechanism

In the basic method, the object locating process simply consists of accessing the local ID Table for the given ID and extracting the corresponding IDTE of the desired object from the ID Table. As discussed in Section 7.2.1, due to the specific replication policy of IDTEs at different nodes of the system, this method does not require any network communication and the object locating operation is performed locally. All the node numbers present in the replica list field of the extracted IDTE are the locations of the desired object's replicas. Therefore, depending upon the type of access operation (READ/UPDATE), the client's access request is sent to one or all of these nodes. When the request can be serviced by accessing a single replica, it becomes necessary to select one node from the replica list in order to send the access request. In our method, the node number whose value is nearest to that of the client's node is selected for servicing the access request. This selection policy is based on the assumption that node numbers are assigned in our system according to the network topology. That is, the difference between the values of the node numbers of distant nodes is larger than the difference between the values of the node numbers of nearby nodes. Sufficient gaps are left between node numbers at the time that the node number values are assigned to various nodes, in order to facilitate future expansion of the network.

It may be noted here that the replica list in the IDTE contains only the list of node numbers as the location of the replicas. Although this information is sufficient to send the access request to the desired object's node, it is not sufficient to access the object, since the object's physical device address (memory location, disk address, etc.) is not yet known. Thus at each node in GALAXY, a separate *local object table* is maintained, which contains information on the physical device addresses of all the objects present on that node. In particular, a local object table contains the following information for each local object: the object's ID and the physical device address of the object. In order to facilitate efficient searching and management of the local object table, instead of maintaining a single local object table for all the objects on a node, a separate local object table is maintained for each type of object that resides on that node, and each local object table is managed by the relevant object manager of that object type. Therefore, once the access request reaches the desired object's node, the relevant object manager searches the local object table corresponding to the type of the desired object to find the physical device address of the desired object.

7.2.3 Advantages of ID- and ID Table-Based Locating

The use of IDs and the ID Tables in the object locating process has the following advantages over the conventional methods, which use direct mapping from names to physical locations of the objects such as node number, inode, and disk address:

1. *Uniqueness*: An ID identifies an object uniquely in the entire system. Unlike external name, the same ID is never used to identify more than one object during the entire life of the system (see Section 5.1). Therefore, even if an object is removed, its ID will not be used again to identify some other object. This avoids the problem of accessing different objects with the same ID at a different time or at a different node.

2. *Flexibility*: IDs are system-wide unique and valid, and do not change even when the physical location of objects changes. Thus, if objects are migrated in the system, there is no need to update the name caches or directories. Only ID Tables must be updated. Thus, the multicache consistency problem is simplified to the consistency of ID Table entries.

3. *Efficiency*: An ID Table is like a cache, which makes the locating operation very fast for objects whose ID is known. Moreover, it contains information about the object's replica locations and its access control information. Thus the accessibility of the object can be verified at the local node, and the nearest replica can be chosen if accessing is permitted. This helps to improve the overall efficiency of the system by reducing the network traffic.

4. *Uniformity*: An ID Table is a mapping from an object's ID to its replicas that contains the same type of information on all the objects in the system, irrespective of their types. Thus the use of IDs and ID Tables facilitates the design of a uniform object management mechanism that simplifies several design aspects of the system, such as consistency control, object migration, object replication, and access control.

GALAXY is not the first system to make use of system-wide unique identifiers in distributed systems. Other systems such as Distributed Smalltalk [2, 8] and Jasmine [31] also use system-wide unique identifiers to identify their objects uniquely in the entire system. However, GALAXY's design is unique in the sense that it facilitates direct locating of all the replicas of the desired object from the client node of the system by using the object's ID and the ID Table. For example, in Jasmine, objects are located by one of two methods: expanding ring broadcast or expanding ring multicast [31]. Both these methods require querying from all the nodes for the desired object in the worst case. Thus the reliability and efficiency of the object locating operation of Jasmine is very poor. On the other hand, Distributed Smalltalk uses the concept of proxy Object and Remote Object Table for its object locating mechanism. Howeve, it does not deal with object replicas and discusses the locating of only one copy of an object. In GALAXY, the nearest replica location of the desired object is always supplied to the user as a result of the object locating operation.

7.2.4 Cost of Updating ID Table

In GALAXY, an IDTE needs to be updated when an object migrates from one node to another, or when an object is replicated at a new node, or when an object is deleted.

When an object migrates from node A to node B, the IDTEs corresponding to this object must be updated to reflect the object's new location in order to facilitate the direct locating of the migrated object. The cost of this update operation basically depends on the number of copies of the concerned object's IDTE at the time of object migration. If there are m copies of the concerned IDTE, then, excluding the local copy, an update message must be sent to the remaining $(m-1)$ copies from node A. Thus the total cost of updating, including the reply messages, becomes $2(m-1)$ messages.

In GALAXY, the number of copies of a particular IDTE basically depends upon the usability of the corresponding object from the various nodes of the system. In the worst case, when the concerned object's usability encompasses all the n nodes of the system, then $m=n$ and the total update cost is $2(n-1)$, where n is the total number of nodes in the system.

However, in a large distributed system, not all the objects are of equal importance to all the users of the system. In fact, in such a system, a particular user normally has access permission for only a very small subset of the set of objects available in the system. Moreover, in a large distributed system, a user normally uses only a very small subset of the set of nodes available in the system. These observations indicate that the usability of a particular object is normally limited to a very small set of nodes in the system, with few exceptions. Thus the value of m for most of the objects will be very small in comparison with n. Therefor, the cost of updating IDTEs will also be normally small for most of the objects. In addition, we have seen in Section 2 that GALAXY assigns IDs only to very large objects such as processes, files, nodes, and devices. These objects are not moved as frequently as fine-grained objects in pure object-oriented systems. Thus the update operation of IDTEs does not occur so frequently in GALAXY. These arguments indicate that the cost of updating the ID Table is also low in the replication or deletion of an object. Thus the management overhead of the ID Table is not very large.

7.2.5 Variations of the Basic Object Locating Mechanism and their Evaluation

Several variations of the basic object locating mechanism were conceived and evaluated. These mechanisms are discussed below, along with the results of their evaluation. The following parameters have been used for the purpose of evaluation:

N=total number of nodes in the system
L=cost of searching the local object table
G=cost of searching the ID Table
M=cost of network communication
f=probability of existence of an object in the creating node
g=probability of existence of an object in the accessing node

The locating cost is defined as the cost of determining the location of the node in which one of the replicas exists.

(1) Basic Method

In this method, the location of the desired object is directly obtained by referring to the ID Table. Thus, the locating cost is simply given by

$$C_1=G$$

(2) Broadcasting

The broadcasting method is included here only for comparison with other methods. In this method, the local object table is first searched, and if the desired object is not found there, a request is broadcasted. Hence the cost of object locating for this method is given by

$$C_2=Lg+\{L+(N-1)(L+2M)\}(1-g)$$

(3) Searching the local object table first

In this method, the local object table is first searched for the desired object. If the object is not found in the local object table, the ID Table is searched to determine the nodes in which the replicas of the object exist. The cost of this locating mechanism is given by

$$C_3=Lg+(L+G)(1-g)$$

(4) Using the number of the creator node for hints on locating

In this method, if the desired object is not found in the local object table, the object table of the node that created the object is searched. If both searches fail, that is, if the desired object's ID is in neither the local object table nor the object table of the creating node, then the usual object locating procedure is performed by accessing the ID Table. The cost of this locating method is given by

$$C_4=Lg+(2L+2M)f(1-g)$$
$$+(2L+gM+G)(1-f)(1-g)$$

(5) Using the number of the creator node for hints on locating and broadcasting

In this method, if the desired object is not found in the local object table, the object table of the node that created the object is searched. If both the searches fail, that is, if the desired object's ID is in neither the local object table nor the object table of the creating node, then the request is broadcasted. All other nodes then access their local object tables and reply. Basically, this method does not require an ID Table. The cost of this locating method is given by

$$C_5=Lg+(2L+2M)f(1-g)$$
$$+\{(N-2)(L+2M)+2L+2M\}(1-f)(1-g)$$

7.2.6 Evalution

Figure 9 shows a graph comparing the object locating mechanisms discused above. It will be observed that the costs of methods (2) and (5), which use the broadcast mechanism, are larger than that of method (3), represented by line C_3, independent of the values of g and N, on the assumption that the message transfer cost (M) is much larger than the cost of searching the local object table (L) or the cost of searching the ID Table (G). Since this assumption is reasonable in distributed systems, broadcast methods (2) and (5) are inefficient. Lines C_3 and C_4 indicate that method (4) is always less efficient than method (3) on the above assumption.

Consequently, searching the local object table first is the most effective of the above mechanisms, but is not always efficient. The graph, shows that the basic method (1) can be more effective than method (3) if the probability of the existence of an object in an accessing node $<L/G$. When L is much smaller than G, and when it is highly probable that the desired object exists in the local node, method (3) (searching the local object table first) is more efficient than the basic method (1). However, the desired object does not always exist in the local node, for example, when files are accessed from a diskless node. In this case, it is preferable to use the basic method.

As can be seen from the graph in Fig. 9, since method (3) performs better than method (1) for a wider range of objects under the assumed conditions, it has been selected as the object locating mechanism of GALAXY. That is, in order to locate any object, GALAXY first searches the local object table to determine its physical device address. If the object is not found there, then the object is not located on the local node. In the next step, the ID Table is searched to determine the physical location (node number) of the object, and then the object table at the node so obtained is searched to determine the physical device address of the object concerned.

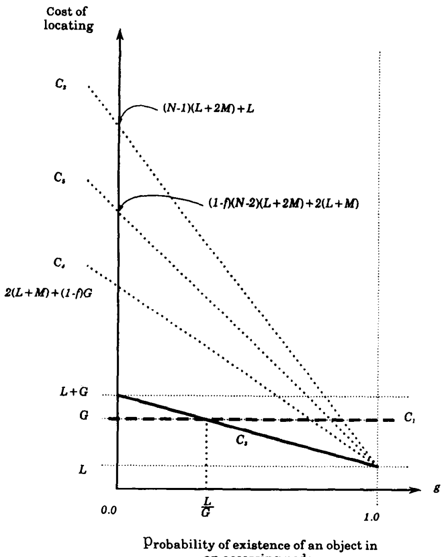


Fig. 9 Comparison of possible object locating mechanisms for GALAXY.

8. Other Implementation Issues of Naming in GALAXY

8.1 Object Migration

We have seen that in GALAXY, the accessing node directly accesses the node at which the desired object is presently located. However, a problem occurs if the object migrates from a node *A* to another node *B* after a process in node *N* issues an access request for the object and before node *A* accepts it. In this case, if node *A* returns a negative reply to node *N*, the same sequence of operations must be repeated until the object is finally located. To avoid this situation, a link is maintained from node *A* to node *B* in node *A*. Consequenlty, node *N*'s request is directly forwarded from node *A* to node *B*. The purpose of using links in our method to keep track of an object's location is different from that of the chaining mechanism used in DEMOS/MP [17] in that these links are not maintained for the entire life of the object, but are deleted when all the IDTEs have been updated and node *A* has been notified of this fact. It may also be noted that, unlike in DEMOS/MP, the failure of a node that holds information on the link will not affect the accessibility of any objects except those on the failed node.

8.2 Access Control

There are three fundamental methods for access control:

(1) *3-tuples*—The system maintains a table in which each entry is a 3-tuple of ⟨process, object, permitted operations⟩.

(2) *Access Control List* (*ACL*)—The system associates with each object the pair ⟨process, permitted operations⟩.

(3) *Capability*—The system associates with each process the pair ⟨object, permitted operations⟩.

We now examine the use of these methods in a distributed environment. Since the management of 3-tuples is centralized, they would not be suitable in a distributed environment. In an ACL, every request is first sent to the node at which the object exists, and is then validated to perform the actual operation. This validation can be done at the accessing node when the capabilities are used; in this case, no actual message transfer is necessary if the requested operation is not permitted. On the other hand, it is easier to modify the access rights of an object in ACL. In the capability method, requests for modification or revocation of capabilities must be scattered throughout the whole system.

In GALAXY, each node has a partial copy of the ID Table, which contains information on all the objects that can be accessed from the node. Thus, by placing the ACL in the IDTE, the requested operations can be validated at an accessing node. This technique makes the access control very efficient.

8.3 Fault Tolerance

Each IDTE in GALAXY has a whole list of the locations at which the corresponding object's replicas exist. Therefore, an object is available as long as at least one node in the replica list is in operation. Each entry in the replica list contains information that indicates whether the node is accessible or not. This information is set by the resource manager of GALAXY, which exchanges management information with other nodes. The object manager selects the best of the accessible nodes by using the nearest replica selection policy, discussed in Section 7.2.2. However, any change of the replica list must be made to all copies of the object's IDTE.

As discussed in Section 4.1.1, GALAXY provides a mechanism of *node group* in which the entire system is hierarchically partitioned into small groups of nodes and each node group has a group leader responsible for maintaining information on the nodes belonging to its group. When one of the replica nodes fails, its group leader holds the complete history of access requests sent to the failed node. When a failed node recovers after some time, the group leader re-sends access requests to this node. This technique can also be used to guarantee the consistency of the copy list in an IDTE. Thus, the IDTEs in the failed node are immediately updated to reflect the current locations of the replicas when it recovers.

8.4 Client-Server Connection

A process that accesses an object must have the ID of that object. In inter-node client-server communication, in which a client requests a remote server to perform the operation and a server sends back the result, the server must know the ID of the client in order to send back the result. A simple approach to handling this is to pass the IDTE of the client along with the request message. However, a large overhead is needed to create a new IDTE at the server node, because all the copies of the IDTE must be updated to maintain consistency. Another approach is to create a temporary IDTE at the server node. This IDTE need only be consistent with the IDTE at the client node. This means that the two are temporarily connected. It is deleted when the server sends back the result. When a client is migrated during the connection, the system only needs to update the temporary IDTE at the server node so that this IDTE points to the new client location. The system also performs an operation to maintain the consistency of the ordinary IDTE.

9. Conclusions

We have described object naming and object locating mechanisms for distributed systems which are network-transparent, efficient, flexible, and easy to use. These mechanisms are partially replicated and distributed among nodes to increase performance and reliability. The replication, distribution, and migration of the naming and the locating mechanisms themselves are also controlled and realized by the naming and the locating mechanisms, because these mechanisms are implemented by the use of GALAXY objects that are replicated and treated in a transparent way. We believe that the mechanisms described in this paper will be useful and applicable to other non-distributed/distributed systems.

GALAXY aims to support a network of nodes scattered over a wide geographical area. We will use ISDN as a wide-area network and are implementing the mechanisms of object naming and object locating in a widely distributed environment of this type. The hierarchical node group concept described in Section 4.1.1 is the key to realizing this.

Acknowledgements

We gratefully acknowledge the technical comments of the anonymous reviewers of the early version of this paper. Their comments have been invaluable in improving the organization, clarity, and contents of the paper.

References

1. Andrews, G. R., Schlichting, R. D., Hayes, R. and Purdin, T. D. M. The Design of the Saguaro Distributed Operating System, *IEEE Trans. Softw. Eng.*, **13**, 1, (1987), 104-118.
2. Bennett, J. K. The Design and Implementation of Distributed Smalltalk, *Proc. OOPSLA '87* (1987), 318-330.
3. Brownbridge, D. R., Marshall, L. F. and Randell, B. The Newcastle Connection, *Softw. Pract. and Expr.* **12**, 12 (1982), 1147-1162.
4. Cheriton, D. R. The V Distributed System, *Comm. ACM*, 31, 3 (1988), 314-333.
5. Cheriton, D. R. and Mann, T. P. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance, *ACM Trans. on Computer Systems*, 7, 2 (1989), 147-183.
6. Comer, D. and Murtagh, T. P. The Tilde File Naming Scheme, *Proc. 6th Int. Conf. Distributed Computing Systems* (1986), 509-514.
7. Comer, D. and Peterson, L. L. A Model of Name Resolution in Distributed Systems, *Proc. 6th Int. Conf. Distributed Computing Systems* (1986), 523-530.
8. Decouchant, D. Design of a Distributed Object Manager for the Smalltalk-80 System, *Proc. OOPSLA '86* (1986), 444–452.
9. Jia, X., Nakano, H., Shimizu, K. and Maekawa, M. Highly Concurrent Directory Management in GALAXY Distributed Operating System, *Proc. 10th Int. Conf. Distributed Computing Systems* (1990), 416-423.
10. Keeffe, D., Tomlinson, G. M., Wand, I. C. and Wellings, A. J. *PULSE: An Ada-Based Distributed Operating Systems*, Academic Press Inc. (1985).
11. Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L. and Stumpf, B. L. The Architecture of an Integrated Local Network, *IEEE Journal on Selected Areas in Communication*, SAC-**1**, 5 (1983), 842-857.
12. Maekawa, M., Kawachiya, K., Ohta, M., Hamano, J. and Shimizu, K. Object Management and Address Space of GALAXY Holonic Operating System, Tech. Rep., Dept. of Info. Sci., Univ. of Tokyo, TR86-15 (1986).
13. Morris, J. H. ANDREW: A Distributed Personal Computing Environment, *Comm. ACM*, **29**, 3 (1986), 184-201.
14. Nelson, M. N., Welch, B. B. and Ousterhout, J. K. Caching in the Sprite Network File System, *ACM Trans. Computer Systems*, **6**, 1 (1988), 134-154.
15. Olson, R. Parallel Processing in a Message-Based Operating System, *IEEE Software*, **2**, 3 (1985), 39-49.
16. Oppen, D. C. and Dalal, Y. K. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment, *ACM Trans. Office Information Systems*, **1**, 3 (1983), 230-253.
17. Powell, M. L. and Miller, B. P. Process Migration in DEMOS/MP, *Proc. 9th ACM SIGOPS Symp. Operating Systems Principles* (1983), 110-119.
18. Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shar, S. and Yueh, K. RFS Architectural Overview, *USENIX Conference Proceedings*, Atlanta, Georgia (1986), 248-259.
19. Ritchie, D. and Thompson, K. The UNIX Time-sharing System, *Comm. ACM*, **17**, 6 (1974), 365-375.
20. Rowe, L. and Birman, K. A Local Network Based on the UNIX Operating System, *IEEE Trans. Softw. Eng.*, SE-**8**, 2 (1982), 137-146.
21. Sandberg, R., Goldberg, D., Kleinman, S., Walsh, D. and Lyon, B. Design and Implementation of the SUN Network File System. *USENIX Conference Proceedings*, Portland, Oregon (1985), 119-130.
22. Schantz, R. E., Thomas, R. H. and Bono, G. The Architecture of the Cronus Distributed Operating System, *Proc. 6th Int. Conf. Distributed Computing Systems* (1986), 250-259.
23. Sheltzer, A. B., Lindell, R. and Popek, G. J. Name Service Locality and Cache Design in a Distributed Operating Systems, *Proc. 6th Int. Conf. Distributed Computing Systems* (1986), 515-522.
24. Shimizu, K., Maekawa, M. and Hamano, J. Hierarchical Object Groups in Distributed Operating Systems, *Proc. 8th Int. Conf. Distributed Computing Systems* (1988), 18-24.
25. Shoch, J. Inter-Network Naming, Addressing, and Routing, *Proc. IEEE COMPCON '78 Fall* (1978), 72-79.
26. Tanenbaum, A. S. and Renesse, R. Distributed Operating Systems, *ACM Comput. Surveys*, **17**, 4 (1985), 419-470.
27. Terry, D. B. Caching Hints in Distributed Systems, *IEEE Trans. on Softw. Eng.*, SE-**13**, 1 (1987), 48-54.
28. Walker, B., Popek, G., English, R., Kline, C. and Thiel, G. The LOCUS Distributed Operating System, *Proc. 9th ACM SIGOPS Symp. Operating Sytems Principles* (1983), 49-70.
29. Watson, R. Identifiers (Naming) in Distributed Systems, *Lecture Notes in Computing Science: Distributed Systems—An Architecture and Implementation*, B. W. Lampson et al. (eds.), Springer-Verlag (1981), 191-210.
30. Welch, B. and Ousterhout, J. K. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System, *Proc. 6th Int. Conf. Distributed Computing Systems* (1986), 184-189.
31. Wiebe, D. A Distributed Repository for Immutable Persistent Objects, *Proc. OOPSLA '86* (1986), 453-465.

(Received March 7, 1990; revised September 28, 1990)