Exploiting Location Awareness for Scalable Location-Independent Object IDs

Gerco Ballintijn Maarten van Steen Andrew S. Tanenbaum

Vrije Universiteit, Department of Mathematics and Computer Science,
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
gerco@cs.vu.nl steen@cs.vu.nl ast@cs.vu.nl

Keywords: wide-area location services, mobile computing, object identifiers, scalability

Abstract

We are building a wide-area location service that tracks the current location of mobile and replicated objects. The location service should support up to 10^{12} *objects on a worldwide scale. To support this huge number of objects, the workload of the location service is distributed over multiple hosts. Our load distribution method is unique in that it is aware of the (geographical) location of the hosts it uses. By using this location knowledge when distributing the workload, the distribution mechanism enforces locality of operations in the location service. Enforcing locality minimizes the use of global network resources by the location service and thereby enhances its scalability. We also show how this location-aware load distribution mechanism can be implemented.*

1 Introduction

Objects provide an easy way to model both applications and system services. It is therefore easy to understand that the use of objects as a design and implementation method has become popular, for example in CORBA [**?**]. Important features object-based distributed systems should have are support for replication and mobility. Replication is used to increase performance and fault tolerance. Mobility has become increasingly prominent, both in hardware (for instance mobile phones and laptops) and in software (for instance mobile agents).

When a client process wants to contact a distributed object, it usually needs to know where the distributed object is. Making this location part of an object reference is problematic for two reasons. First, encoding locations makes sense only if objects hardly or never move. This is generally unrealistic. Second, a replicated object may reside at several locations. To allow a client to locate, say the nearest replica, requires that *all* locations are stored in the object reference.

A location service can be used to support object replication and mobility. The task of a location service is to track the current set of locations where an object resides. A client process can query the location service to obtain the most current set of locations of the object. As part of our research on a worldwide distributed system called Globe [**?**], we are building a wide-area location service. Our current goal is a location service that supports a worldwide distributed system with in the order of 10^9 users and 10^{12} (distributed) objects.

A centralized location service is clearly impossible, given the sheer number of distributed objects. Forms of load distribution are therefore needed throughout the location service. In addition, we also want to minimize the usage of network resources, by localizing processing as much as possible. The main contribution of this paper is that we describe how a worldwide location service can be made scalable by distributing and localizing workload.

The rest of this paper is structured as follows. Section 2 describes how naming is done in Globe, and gives the general architecture of our location service. Section 3 describes how we distribute the workload within the location service, followed by Section 4 that shows how we can minimize the network usage by localizing the distributed workload. Section 5 describes how our ideas can be implemented. Section 6 describes related work, and in Section 7 we draw our conclusions.

2 A Wide-Area Location Service

In our model, a **contact address** specifies **where** and **how** to contact a distributed object. An example of a contact address is a URL. It consists of a scheme identifier that specifies the communication protocol (how) and an address (where) related to the scheme. The relationship between a distributed object and its contact address is transient, since the object can move to another host and the contact address can be reused by other objects.

Replication and mobility have a significant impact on the relationship between objects and contact addresses. Replication implies that an object can be contacted at multiple locations. A single object can thus have a *set* of contact addresses. Since objects are allowed to change location, the set of contact addresses of an object can change frequently.

Naming services like the Internet Domain Name System (DNS) [**?**] and the X.500 Directory service [**?**] are traditionally used to provide this kind of object-to-address mapping. Unfortunately, services such as these assume a relatively stable object-to-address mapping to enable efficient implementations. Given our desire to support mobility, a different solution is needed.

2.1 Naming Architecture

To support highly mobile objects, we use separate **naming** and **location** services, and introduce **object handles**. An object handle uniquely identifies a distributed object, throughout the object's *entire* life time. The object handle is location independent, since it is not allowed to change when the object changes its location. The naming service binds user-friendly (e.g. ASCII) names to an object handle. The location service maps an object handle to a set of contact addresses. Finding an object consists of two phases, using a naming service to find the object's object handle, and using the location service to find the object's current set of contact addresses.

The location service provides three basic operations: look-up, insert, and delete contact addresses for a given object. Its primary function is to look up (some of) the contact addresses of an object handle. The insert and delete operation are referred to as update operations.

2.2 Location Service Structure

To implement efficient look-up and update operations, our wide-area location service partitions the underlying network into a hierarchy of (geographical) domains (see Figure 1). At the top of the hierarchy is the root domain that comprises the whole network. At the bottom of the hierarchy reside the leaf domains. Leaf domains consist, for instance, of a few interconnected LANs. Associated with every domain is a directory node. A directory node stores location information for the objects within its associated domain. The directory nodes together form a distributed search tree.

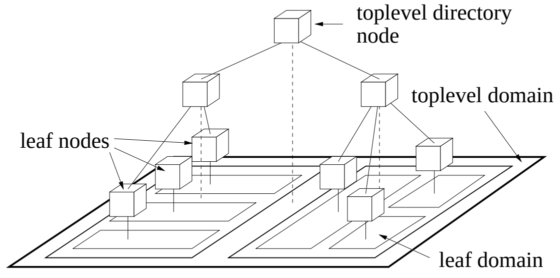

Figure 1: Hierarchical network partition

A directory node associates a **contact record** with every known object handle. The contact record stores the location information of the object handle. A contact record stores either contact addresses from the domain of the the record's directory node or **forwarding pointers**. A forwarding pointer points to a child node (subdomain) of the node containing the forwarding pointer. The forwarding pointer indicates that contact addresses of the object handle can be found in the subtree rooted by the child node. Every contact address can be found by following a path of forwarding pointers from the root node down to its leaf node.

Figure 2 shows as an example the contact records for one object handle. In this example, root node N0 has one forwarding pointer for the object handle, indicating that contact addresses can be found in its right subtree, rooted at node N1. Node N1, in turn, has two forwarding pointers, pointing to nodes N2 and N3, where the actual contact addresses are stored. To simplify the discussion we assume that contact addresses are always stored in leaf nodes. It is, however, also possible to store contact addresses at intermediate nodes [**?**].

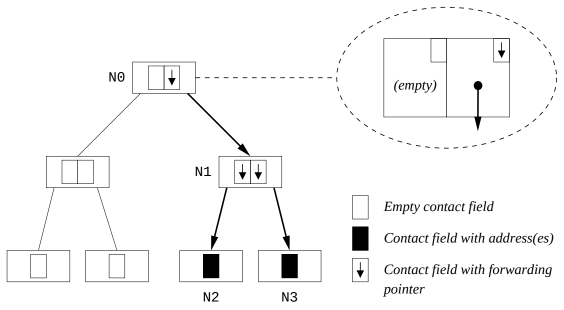
N0
(empty)
N1
N2
N3
Empty contact field
Contact field with address(es)
Contact field with forwarding pointer

Figure 2: Example: search tree for one distributed object

When a client wants to know the contact address of an object, it initiates a look-up operation at the leaf node in the domain in which it resides. The client provides the object's object handle as parameter. The look-up operation starts by checking if the leaf node has a contact record for the object handle. If the leaf node has a contact record, the operation returns the contact address found in the contact record. Otherwise, it recursively checks nodes on the path from the leaf node to the root. If the look-up operation finds a contact record at any of these nodes, the path of forwarding pointers starting at this node is followed downwards to a leaf node where a contact address is found. If no contact record is found at any of the nodes on the path from the leaf node to the root, the object handle is unknown.

The goal of the insert operation to store a contact address and create a path of forwarding pointers to the contact address. When an object has a new contact address in a leaf domain, the object inserts this new contact address at the node of the leaf domain. The insert operation starts by inserting the contact address in the contact record of the leaf node. The insert operation then recursively requests the parent nodes to install a forwarding pointer. The recursion stops when a node is found that already contains a forwarding pointer, or otherwise at the root. The insert operation that inserts an object handle's first contact address is referred to as the object handle's **initial registration**. The delete operation removes the contact address and path of forwarding pointers analogous to the insert operation. Algorithmic details can be found in [**?**].

3 Load Distribution

We first focus on the workload scalability problem. The root node of the search tree has to store contact records and handle look-up and update requests for all object handles currently in use. This occurs because every contact address of every object needs to be reachable from the root node. Since we want to support 10^{12} objects, the root node will contain 10^{12} contact records. If the size of a contact record is 100 bytes, the storage capacity required is 100 terabytes. The number of accesses to the root node is an even bigger problem. Even if every contact record at the root is accessed only once a year, the root node still needs to able to handle approximately 3.2×10^4 accesses per second.

The solution is to divide the work of a (logical) **tree** node and use multiple **physical** nodes (machines) to handle the workload of the root node. To distribute the load of a tree node efficiently over its physical nodes, we need to fulfill the following requirements. First, every physical node should be able to handle its workload independently of other physical nodes. A dependency between physical nodes implies extra communication, which would introduce extra overhead for operations. Second, it should be easy to transfer and redistribute the workload over a set of physical nodes. This is needed to deal with changes in the usage of the system. When physical nodes are added or removed, the workload needs to be redistributed to adapt to the new situation. Third, it should be easy to determine which physical node handles which part of the total workload. Specifically, a caller should be able to determine *locally*, that is without any further communication, with which physical node to communicate.

To fulfill these requirements, we propose the following solution. The load distribution will use the contact record as a relocatable unit of work. The workload of a tree node is thus the set of all the contact records it stores. Every physical node will handle a subset of this workload. For ease of discussion, we consider the subsets to be disjoint. A special physical node selection field will be added to the object handle. This selection field will be used to determine at which physical node to store a contact record. By choosing the contents of the selection field carefully, we can influence the choice of the physical node to be used by the associated contact record. Note that this field is used only to guide the search within the location service. It has nothing to do with where the object is currently located.

This general architecture fulfills the requirements stated above. Since operations on contact records are independent and the subsets stored by physical nodes are disjoint, the first requirement is easily fulfilled. The contact record is also easily transferable, since it is a simple self-contained data structure, fulfilling the second requirement. By basing the choice of a physical node on the selection field of the object handle, a sender can determine by itself which physical node to contact, fulfilling the third requirement. The actually selection process at the sender should of course be lightweight. Figure 3 shows the contact records of one object handle placed at one physical node in every tree node.

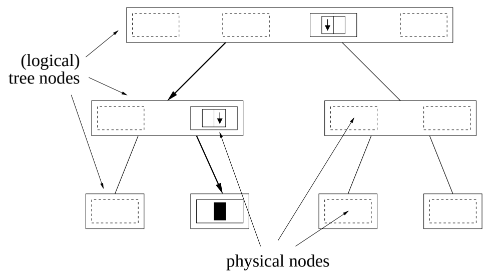

Figure 3: Tree with contact records of one specific object handle

4 Nearby Communication

The discussion in Section 4 does not specify a specific placement strategy. It does not specify which contact record to place on which physical node. In this section, we first show a simple placement strategy and explain what is wrong with it. We then describe our proposed solution and show its improvements.

4.1 Hashing

A naive approach would be to place contact records at physical nodes in a random fashion. This can be done by inserting a random value in the selection field of the object handle. This value can then be used in a hashing scheme. This approach has excellent load balancing characteristics, since it can provide a uniform work distribution. Unfortunately, it has also poor communication patterns. This can be explained by the following scenario.

Consider a search tree with three levels: state, continent, and world (see Figure 4). In the tree, there is an Atlanta leaf node, which consists of just one physical node. Its parent, the U.S. tree node, consists of two physical nodes, one in San Francisco and one in Washington D.C. . The root node consists of a number of physical nodes, one of them located in New York. Now consider what could happen when a new contact address for an object handle is inserted in the Atlanta leaf node. For the logical tree node of the U.S. domain, the object handle hashes to the physical node in San Francisco, and the contact record is thus stored in San Francisco. For the worldwide domain, the object handle hashes to the physical node in New York, and the root contact record is stored there. The insert operation therefore visits nodes in Atlanta, San Francisco, and New York (in that order) to insert the contact address and create a path of forwarding pointers.

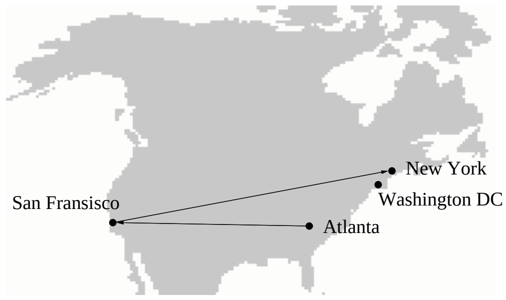

Figure 4: Communication direction while going up in the tree

This example shows a very inefficient communication pattern. We would like to avoid this erratic criss-cross pattern. In fact, we simply want to use the physical node in Washington D.C., not the physical node in San Francisco. In more general terms, we would like to use physical nodes in the general vicinity of Atlanta. To ensure this, we need to augment the load distribution solution with a solution for nearby communication. This brings us to the second scalability problem.

4.2 Forcing Locality

To explain our proposed solution for ensuring nearby communication, we first look at the communication patterns in the tree during an object's initial registration. We then generalize our ideas to include all the possible communication for an object handle.

4.2.1 Registration Communication Pattern

When performing the initial registration of an object handle, we want to store our new contact records (on the path of the leaf node to the root node) at physical nodes that are preferably geographically near to each other. By using physical nodes that are geographically close by, we can avoid using long-distance networks and keep the distance traveled small when traversing the tree. This, in turn, enhances the scalability of our location service.

We make the assumption that a large *geographical* distance between physical nodes implies a large *network* distance. We feel that in current wide-area networks this assumption is generally realistic when talking about large distances in the order of a 1000 km or more. Since this assumption is only a heuristic, there will be exceptions to the rule. We expect, however, that given the increasing prevalence of the networks, this assumption will become valid for smaller distances in the future.

By placing contact records at different levels at physical nodes that are in each other's general vicinity, one creates a kind of virtual column through the tree (see Figure 5). The object's physical root node is the top and the leaf node is the bottom of the column. We therefore want the geographical location of the leaf node to determine the physical nodes used at every tree node on the path from leaf to root. We can do this by encoding the geographical location in the selection field of the object handle. The placement strategy would then be able to place a contact record at the physical node closest to the location in the object handle. The location can, for example, be encoded as the leaf node's longitude and latitude.

4.2.2 General Communication Patterns

The column notion is specific to the initial registration of an object. The notion of avoiding the criss-cross communication pattern is, however, more general, and should apply to all communication. The vicinity requirement can be generalized informally by saying that the communication between levels in the tree should at least not switch geographic direction when communicating at longer distances. If a leaf

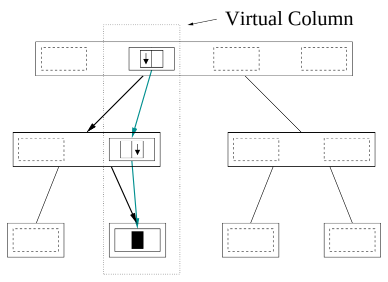

Figure 5: Column for one object handle in a partitioned tree

node and physical root node are far apart, then the path traveled while going higher in the tree should always go in the same general direction (see Figure 6). By going up one level in the tree the physical node that stores the contact record should either be in the general vicinity of the calling (child) node, or be closer to the object's physical root node.

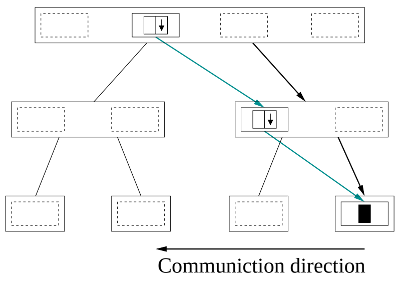

Figure 6: Communication direction while going up in the three

As in the hashing example above, going to San Francisco from Atlanta, and coming back to New York is not efficient. If the address was inserted in Los Angeles, the pattern Los Angeles, San Francisco, New York would be acceptable, since the general direction does not change. The resulting requirement on physical nodes used can be depicted as a pyramid-like shape. The top of the pyramid can still be the geographical location of the leaf node used for the initial registration of the object handle.

Figure 7 shows an example of the pyramid shape. The center of the pyramid is determined by the object handle. The grey squares represent the physical nodes used by the object handle. The tree has three levels: the root, intermediate, and leaf level. The root level has one tree node consisting of sixteen physical nodes. The intermediate level has four tree nodes, each consisting of four physical nodes. The leaf level has sixteen unpartitioned leaf nodes. At the root level only one physical node is used to store the object handle's associated contact record. At the intermediate level, every tree node has one physical node that stores the contact record. The physical nodes are located close to the physical node at the root level. At the leaf level, no partitioning is used, so every leaf node will store a contact record of the object if appropriate.

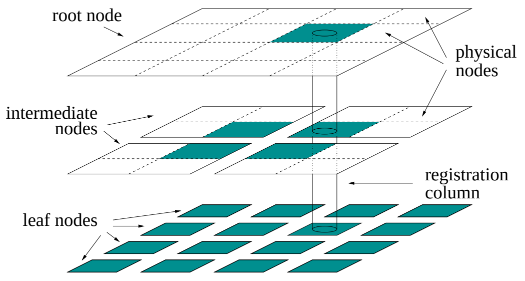

Figure 7: Pyramid for one object handle in a partitioned tree

Adding a location to the object handle does not endanger the object handle's location independence. The object handle can still be used to insert contact addresses at every leaf in the tree. From the (logical) tree's viewpoint, the location is just some random bits of the object handle.

5 Implementation

The design of a physical node can be divided into three layers: the algorithm, selection, and communication layer. The algorithm layer contains the implementation of the update and look-up operations. The operations use only the logical search tree and have no knowledge of node partitioning. They are implemented using the RPC primitive. The selection layer provides an RPC interface to the algorithm layer. It contains, however, only the code responsible for selecting the proper physical node. The communication layer is responsible for the actual communication. The selection layer takes a **tree node identifier** and an object handle and converts those to a **physical node identifier**. The communication layer implements the RPC semantics by exchanging messages with the physical node selected by the selection layer. The communication layer is responsible for resolving physical tree node identifiers to network addresses.

The selection layer works *conceptually* as follows. When an operation in the algorithm layer needs to communicate with, for instance, the parent, the operation invokes the RPC primitive provided by the selection layer. The selection layer computes for every physical node of the parent its distance to the location in the object handle. The selection layer then selects the physical node with shortest distance to the location, and subsequently initiates a message exchange at the communication layer. To compute the distances the selection layer maintains the set of physical nodes of the tree nodes with which it will communicate, for instance, the parent and child nodes.

5.1 Requirements

The location-based selection method has to fulfill certain requirements.

R1 The selection process should be deterministic and unique. As long as the tree does not change, the same physical node should be returned. Moreover, to avoid ambiguity only one physical node should be returned. It is inefficient to have to check multiple physical nodes.

R2 The location information should be durable. Since objects are allowed to be long lived, we can expect object handles to have a longer live span than a single configuration or even implementation of the location service. The location information should therefore be usable in different configurations of the tree and across new versions of the location service.

R3 The third requirement is that the location information should use a small number of bits in the object handle, as object handles are used as general references in our system.

R4 The selection process should be fast and thus local. Since this process is on the critical path, it should take as little time as possible.

R5 It should be easy to add, remove, or move physical nodes. Since we can imagine the logical tree and its partitioning being adapted regularly to suit the current situation, these modifications should not require much work or have a large impact on the tree as a whole.

R6 The storage and communication overhead introduced by partitioning and selecting a physical node should be reasonable.

5.2 General Implementation

Conceptually, a node recomputes the distance to the same or a similar location every time the location is used in communication. If we consider that the root node might have on the order of 10^3 or 10^4 physical nodes, computing all distances is clearly undesirable, given requirement R4. We can, however, take the distance computation step out of the critical communication path, by creating a *location-mapping table* off-line and using the location as an index in this table.

We create the location-mapping table, as follows. We divide the surface of the earth into a large number of small disjoint elementary areas. This division is, in principle, independent of the partitioning used by the search tree, but will, in general, be similar. If a specific tree node N has been partitioned into physical nodes $PN_1, \ldots, PN_k$, we assign PN_i to elementary area A if PN_i is in, or closest to A. Each tuple (A, PN_i) forms an entry in the mapping table of node N. The mapping table of node N is distributed to all physical nodes that may need to communicate with node N. When a physical node is added to or removed from the set of physical nodes of node N, all mappings of node N need be recomputed and distributed again. A versioning scheme is needed to ensure that caller and callee use the most up-to-date version of the mapping table.

5.3 Naive Implementation

A straightforward way to create elementary areas is by creating a grid on the earth's surface using longitude and latitude. The longitude ranges from 180° west to 180° east, and the latitude ranges from 90° north to 90° south. If we use, for example, $1° \times 1°$ degree areas, this results in 64800 elementary areas. The (longitude,latitude) coordinate of these areas can be used as a location data structure. We implement the mapping table using a 2-dimensional array. The (x,y) coordinate is the index of the array.

This implementation fulfills most requirements easily. The mapping table ensures that the selection process is deterministic and unique, fulfilling requirement R1. Longitude and latitude values are stable and thus fulfill requirement R2 (durability). The third requirement (size of location information) depends heavily on the resolution (size of an elementary area) used. In the example above the size is 17 bits. Requirement R4 (fast execution) is fulfilled by using an efficient table-indexing operation. Since adding or removing a physical node simply requires recomputing and redistributing the mapping table, requirement R5 is easily met. Meeting requirement R6 depends, just like R3, heavily on the resolution used. If we use a 4-byte physical node identifier, the example above gives tables the size of $64800 \times 4 = 253$ kilobytes.

There are two kinds of problems with this naive solution. First, if we want a to use a higher resolution for our location information the table size increases dramatically. For instance, if we increase our resolution to elementary areas $0.1° \times 0.1°$ the size of the mapping table becomes $6.5 \times 10^6 \times 4 = 25$ megabytes. Given that tree nodes might have in the order of 100 to 1000 children, this implementation requires 2.5 to 25 gigabyte of main memory. This implementation also requires large network resources, since mapping tables are distributed regularly. The naive implementation can thus support only a limited resolution.

The second problem is the inefficient use of table space. There are several reasons for this. If we consider sparsely populated areas like oceans and deserts, it is clear that we do not need the same kind of resolution at every location on the surface of the earth. Another source of inefficiency is that the actual size (in

km^2) of an elementary area differs across the earth. Since we use a Mercator-like projection, there are more elementary areas per km^2 near the north and south pole than at the equator. Also, large parts of the mapping table will contain the same physical node identifier. Consider a small domain with only a few physical nodes. The location-mapping table for this domain will have a large number of locations which map to the few physical nodes that comprise the domain. This ratio becomes even worse when using a higher resolution.

5.4 Mapping-Table Compression

The basic problem of the naive implementation is the large size of the mapping table. The size leads to large main-memory and communication requirements threatening requirement R6. If we want to support higher resolutions, we need to implement a smaller mapping table. The large size is the result of using a (two-dimensional) array to implement the mapping table. The array contains large parts storing the same physical node identifier. We want to compress these parts. However, we still want to have a fast indexing operation on the mapping table.

We can use a quadtree [**?**] to implement a smaller mapping table. The quadtree represents the hierarchical partitioning of the earth's surface. Instead of dividing the surface of the earth per degree, the surface is repeatedly partitioned in four equally sized smaller parts. The top level surface of $360° \times 180°$ thus contains four $180° \times 90°$ parts, which in turn contain four $90° \times 45°$ parts, etc. (see Figure 8). The partitioning stops at the level of elementary areas, for instance areas of approximately $1° \times 1°$. Leaf nodes of the quadtree represent elementary areas, and store the physical node identifier associated with their elementary area. Using the mapping table to obtain the physical node identifier of a location consists of traversing the quadtree until a leaf is reached.

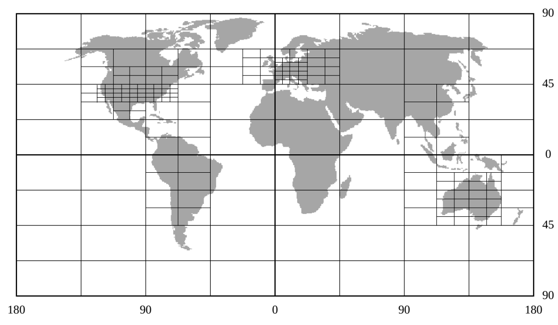

Figure 8: Quadtree covering the earth's surface

We compress the information in the mapping table by not building the complete quadtree to the elementary area level. If all the leaf nodes in a subtree store the same physical node identifier because they are all assigned to the same physical node, only the root node (storing the physical node identifier of its leaf nodes) needs to be created. The height of the quadtree thus depends on the level of detail required at a certain area. If we consider that 70% of the surface of the earth consists of water, there are a considerable number of subtrees that cover oceans and seas. All these subtrees are likely candidates for compression.

6 Related Work

Most existing location systems can be divided into three categories: (traditional) name servers, home-based approaches, and systems using forwarding addresses.

Well known systems in the name server category are the Internet's Domain Name System (DNS) [**?**], DEC's Global Name Service (GNS) [**?**], and the X.500 Directory service [**?**]. These systems achieve scalability through load distribution and server replication. Load distribution is achieved by distributing parts of their name space over different servers. Servers are in turn replicated to increase their availability. These systems make the assumption that the name-to-address binding is relatively stable. We cannot make this assumption, if we want to support highly mobile objects. The systems provide also not complete location independence, since since resolving a name means visiting several servers.

Current designs for location services in Personal Communication Systems (PCS) range from single-level home-based approaches to hierarchical solutions like ours. In the home-based approach, each object has a designated server, called its home, that keeps track of the object's current location. To locate an object, we need to contact the object's home to find out the actual location. Obviously, home-based approaches cannot scale. The home-based approach is also used by mobile IP [**?**].

Some improvement is made by introducing more levels. In particular, most PCS location services use a two-level scheme in which the local server is contacted first, and in the case of failure, contact is made with the home. Proposals for several levels have also been introduced [**?**, **?**]. Apart from functional differences with our approach, none of these systems address worldwide scalability as discussed in this paper. In particular, node partitioning and load balancing is not considered.

Two systems in the forwarding addresses category are Location Independent Invocation (LII) [**?**] and Stub-Scion Pair (SSP) Chains [**?**]. These systems use a forwarding address as the basis for their distributed object references. When an object moves from one host to the next, it leaves a forwarding address. Objects are found by following the chain of forwarding address. Since no centralized component is used in locating objects (in principle), the workload is evenly distributed. The systems are, however, vulnerable to erratic communication patterns, since no locality is used. When an object moves frequently across large distances, following the chain of forwarding references will require much communication. The LII systems has an additional problem in that it uses a name server when following the chain of forwarding addresses fails.

The use of quadtrees in our implementation of the location-mapping table is similar to other spatial data structures. Building a tree by recursively dividing up a space, is a well-known method to efficiently parallelize applications, as used by Multi-Grid methods like Barnes-Hut [**?**]. Samet describes in [**?**] an image compression technique using quadtrees that is similar to our table compression.

7 Conclusion

In this paper, we have described a unique approach in using location awareness to increase the scalability of a wide-area location service. Using (geographical) location awareness allows the nodes in our location service to reason about distances and thereby avoid erratic crisscross communication patterns. We have also described how such ideas can be implemented efficiently using a location-mapping table.

We are currently implementing the location-aware load distribution in our location service prototype. This will allow us to experiment and validate our ideas. Other current and future work consists of finding better ways to implement location-mapping tables, and more general using locality where possible.

References

[1] OMG. The Common Object Request Broker: Architecture and Specification, revision 2.2. Technical report, Object Management Group, Feb. 1998.

[2] M. van Steen, P. Homburg, and A. S. Tanenbaum. The Architectural Design of Globe: A Wide-Area Distributed System. Scheduled for publication in IEEE Concurrency, 1999.

[3] P. Mockapetris. RFC 1034: Domain Names - Concepts and Facilities, Nov. 1987.

[4] S. Radicati. *X.500 Directory Service: Technology and Deployment*. International Thomson Computer Press, London, 1994.

[5] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 104–109, Jan. 1998.

[6] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic Design of the Globe Wide-Area Location Service. *The Computer Journal*, 41(5):297–310, 1998.

[7] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[8] B. Lampson. Designing a Global Name Service. In *Proc. 4th ACM Symposium on Principles Of Distributed Computing*, pages 1–10. ACM, 1985.

[9] C. Perkins. IP Mobility Support. RFC 2002, Oct. 1996.

[10] R. Jain. Reducing traffic impacts of PCS using hierarchical user location databases. In *Proc. IEEE Intl. Conf. Comm.*, 1996.

[11] J. Wang. A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems. *IEEE J. Selected Areas Commun.*, 11(6):850–860, 1993.

[12] A. Black and Y. Artsy. Implementing Location Independent Invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119, 1990.

[13] M. Shapiro, P. Dickman, and D. Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.

[14] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[15] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.