Achieving Scalability in Hierarchical Location Services

Maarten van Steen
Gerco Ballintijn
Vrije Universiteit, Amsterdam
{*steen,gerco*}*@cs.vu.nl*

Internal report IR-491
January 2002 (revision)

Abstract

Services for locating mobile objects are often organized as a distributed search tree. A potential problem with this organization is that high-level nodes may become a bottleneck, affecting the scalability of the service. A traditional approach to handle such problems is to also distribute the location information managed by a single node across multiple machines. However, combining distribution with exploiting locality is difficult. We introduce a method that radically applies distribution of location information such that the load is evenly balanced across *all* machines that form part of the implementation of the service, while at the same time exploiting locality. Our method is largely independent of the usage and migration patterns of mobile objects. We demonstrate that it can scale better than the traditional home-based approach.

Keywords: mobile computing, location services, scalability, performance evaluation, wide-area systems

vrije Universiteit

Department of Mathematics and Computer Science

1 Introduction

Efficiently locating and tracking (mobile) objects is important in any distributed system. With the continuous expansion of the Internet and in particular the increase of the number of mobile devices, we need solutions that can efficiently work in small-scale distributed systems but that can also scale and sustain as these systems eventually expand across larger networks and support more objects.

Location services that are organized as a distributed search tree meet this requirement as they exploit locality for looking up and updating addresses while at the same time are capable of spanning networks the size of the Internet [17]. However, the hierarchical organization of the service suggests that high-level nodes may form a potential performance bottleneck that can severely limit its scalability. The Globe location service is also logically constructed as a tree [24], but avoids scalability problems by distributing the location information stored at a single node across *all* machines that jointly implement the service. The combination of a logical hierarchical organization and this radical distribution of location information leads to an efficient and scalable solution.

In this paper, we present the basic organization of the Globe location service and discuss its implementation by means of a collection of servers running on machines that are spread across a wide-area network. Although we concentrate on Globe, our approach is equally applicable to other location services that assume the underlying network is organized into a hierarchy of domains.

The main contribution of this paper is that we present a method by which performance problems for hierarchical location services can be avoided by properly distributing information across available servers. Our method is unique in the sense that it establishes good load balancing while preserving locality in the execution of lookup and update operations. To substantiate our claims, we conducted a number of simulation experiments using data from the World Cities Population Database as input [18]. We not only show that building scalable and sustainable hierarchical location services is feasible, but also that they can outperform traditional home-based approaches when it comes to scalability.

We organize our discussion as follows. In Section 2 we present the principal working of the Globe location service and briefly relate it to other systems. The core of this paper is formed by Section 3, in which we explain how location information is spread across the collection of servers that implement the service. The results of our simulations are discussed in Section 4. This section also compares the hierarchical approach to the more traditional and widely-applied home-based approaches [14, 15] and we demonstrate that hierarchical solutions are often better. We come to conclusions in Section 5.

2 The Globe Location Service

The Globe location service is representative for many hierarchical location services. In this section, we briefly discuss its organization and concentrate only on the main algorithms that can also be found in similar services. Details on our algorithms as well as various optimizations that make our approach different from others are discussed in [2, 23].

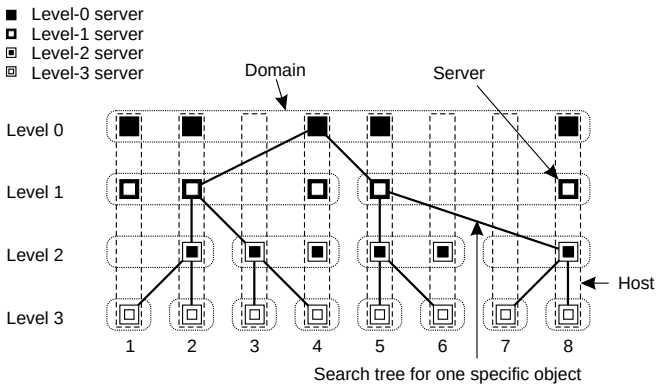


Figure 1: An example of organizing the collection of LS hosts into a hierarchical organization of domains, with each domain having at least one location server.

2.1 General Model

We assume that a (mobile) entity is represented by a single object that has a globally unique, location-independent **object identifier** (**OID**). Each object can be contacted at its **contact address**, which is stored by the location service. When an object moves, it changes its contact address requiring the contact address as stored by the location service to be replaced with the new address.

We assume a location service is implemented by means of several dedicated (nonmobile) processes spread across a network that store and maintain information on the location of an object. Such a process is called a **location server**. A machine hosting a location server is called a **location server host** or simply **LS host**. Location information is stored in a **contact record** and either consists of a contact address or a pointer to another location server. A pointer means that the other location server also stores a contact record for the same object. Cyclic references are not allowed. A location server may store only one contact record for any given object.

We assume there is a nonhierarchical (potentially large) set of LS hosts spread across a network. We organize these hosts into a hierarchical collection of **domains**. The top level, denoted as level 0, consists of a single domain that covers the entire network. Each domain *D* may be partitioned into a next level of smaller **child domains**, turning *D* into their **parent domain**. A lowest-level domain typically corresponds to a campus or a city.

Every domain, regardless its level, is assumed to have at least one associated location server. A server is always associated to one domain, but there may be several servers associated to the same domain. Because a server needs to be hosted by an LS host, it also follows that every lowest-level domain contains at least one LS host. An LS host may be running servers from different domains, as shown in Figure 1.

In our model, when a request related to object *O* needs to be forwarded by a server *S*, it can be forwarded only to the location server pointed to in the contact record for *O* stored by *S*. If *S* has no contact record for *O*, the request is always forwarded to the same location server at the next higher-level domain (but which may be different for different objects). In this way, we guarantee deterministic behavior of lookup and update algorithms. The details of these algorithms can be found in [23].

The problem that we address in this paper is deciding for each domain what the best server is to store the contact record of a given object, or to forward a request to regarding that object. As we explained, for a given object we always choose the same server. Having a hierarchical organization of domains, this effectively leads to the construction of a collection of search trees, one tree *per object*, as also shown in Figure 1. An important observation is that the collection of root nodes of these search trees may be completely distributed across the location servers in the top-level domain.

2.2 Operations

We now describe how lookup and move operations are carried out on a single object O. We take a simplified approach and concentrate only on the essence of these operations. Details can be found in [2, 23].

We use the notation $addr(O)$ to denote the current contact address of object O. Let $D_k(A)$ denote the domain at level k containing address A with $k = 0$ being the top-level domain. $S_k(O,A)$ denotes the unique location server in domain $D_k(A)$ that may store a contact record for object O. We generally omit the first parameter and write $S_k(A)$. For simplicity and without loss of generality, we can assume that all lowest-level domains are at the same level, say n.

Consider an object that registers a new contact address $A = addr(O)$. It contacts the location server $S_n(A)$ in the lowest-level domain containing A, which subsequently stores A. Then, for each level $k > 0$, location server $S_k(A)$ contacts server $S_{k-1}(A)$ in the parent domain and requests it to store a forwarding pointer to $S_k(A)$. The result is that a path of forwarding pointers is created from O's top-level server $S_0(A)$ down to $S_n(A)$.

A move operation in our simplified model consists of a pair of *(insert,delete)* operations. When an object O wants to move from address A to B, it first initiates an insert operation for address B. This address is stored in $S_n(B)$. Analogous to the registration of the first address for an object, each server $S_k(B)$ requests server $S_{k-1}(B)$ in the parent domain to store a forwarding pointer. However, instead of proceeding up until O's top-level server, the insert request is no longer forwarded when it reaches O's server in the smallest domain containing both A and B, say $D_l(A)$. After the insert operation has finished, the delete operation simply removes the path of forwarding pointers from $S_l(A)$ to $S_n(A)$, after which it removes A from $S_n(A)$ completing the move operation.

Looking up an object is relatively simple. Assume we have a client located at address C. The client first contacts $S_n(C)$, that is, the server for O in the leaf domain where the client resides. If $S_n(C)$ does not contain a contact record for O, it passes the lookup request to O's server $S_{n-1}(C)$ in the parent domain. In general, location server $S_k(C)$ passes a lookup request for O to $S_{k-1}(C)$, unless it contains a contact record for O. The first server $S_k(C)$ that stores a forwarding pointer for O then passes the lookup request along the downward path to $S_n(addr(O))$ where the object's current address is stored.

Note that both move and lookup operations exploit locality. An object moving from A to B generates traffic that is passed only between location servers in the smallest domain containing both A and B. Likewise, traffic for a lookup request takes place between only those servers that are in the smallest domain in which both the requester and the object reside.

2.3 Related Work

Location services that are based on a hierarchy of domains have been proposed for next-generation mobile-communication networks and general-purpose distributed systems [1, 4, 5, 9, 10, 12, 16, 25]. Differences between these services are found in the way domains are used and constructed, and the various optimizations to reduce the length of search and update paths.

Hierarchical location services share the problem that a server for a high-level domain may become a bottleneck impeding the scalability of the service. Several solutions have been proposed to reduce the load on servers in high-level domains. One class of solutions comprises the construction of short-cut links to servers in low-level domains. If it is known that an object (generally) resides in low-level domain D, servers in other domains may cache a pointer to D and immediately redirect lookup requests to D, thus avoiding that high-level nodes need to process the request [2, 10]. Standard techniques for purging cache entries are used to keep entries up-to-date.

Another way to offload servers in high-level domains is to install redirection pointers. When an object moves from domain $D_k(A)$ to a same-level domain $D_k(B)$ the server in $D_k(A)$ stores a pointer to $D_k(B)$ [16]. In other words, servers in domains at levels higher than k are not informed about the object's migration. This approach effectively introduces a chain of forwarding pointers between servers in different low-level domains and is comparable to approaches for locating mobile objects in local-area distributed systems [6, 7, 11, 21]. Additional techniques are needed to reduce the length of chains.

Orthogonal to introducing additional pointers is to distribute the load among servers in high-level domains by introducing a fat tree [13]. In this approach, the set of object identifiers is divided into equally-sized subsets, effectively using a hashing scheme based on the m most significant bits of OIDs. Each subset is managed by a separate server. This approach has also been applied in NLS [9] and resembles the number-based routing as applied in peer-to-peer networks [19, 22, 26]. It works fine for systems in which locality is not really an issue, such as the CM-5 supercomputer where it was originally applied, but fails to work efficiently in wide-area systems in which exploiting locality is crucial for scalability.

3 Object-to-Server Mapping

The situation that we need to deal with can also be formulated in terms of the following **mapping problem**. Given a collection of objects and location servers in a domain D, how can we associate each object to a single server, such that this leads to an efficient implementation of lookup and move operations across all domains? Let $map(O,D)$ denote the method that selects a server for O in D. We are looking for an implementation of *map* that meets the following requirements:

R1 $Map(O,D)$ is deterministic and unique: it always returns one and the same server as long as the mapping is not explicitly changed.

R2 Computing $map(O,D)$ is efficient in time and space, meaning that storage, computing, and communication overhead needed to implement *map* are kept to a minimum.

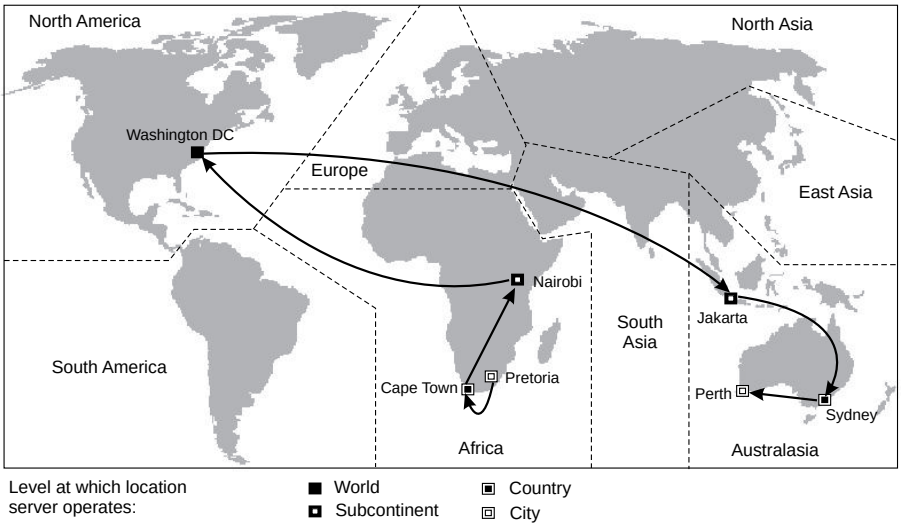


Figure 2: A wrong mapping introduces more communication.

R3 For each domain D and parent domain D', the costs in communication between the server $map(O,D)$ and the server $map(O,D')$ is kept to a minimum.

R4 It is easy to add or remove servers, or to associate an object O to another server in the same domain. In other words, it is easy to change $map(O,D)$.

R5 The number of objects supported by the same server is not too large; we need to avoid that a server becomes overloaded because it has too many contact records.

Mapping an object to an appropriate server and thereby meeting requirement R3 is important. Consider Figure 2 that shows the division of the network into **subcontinents**, which are level-1 domains that span a (relatively large part of a) continent. Each subcontinent is divided into countries, which, in turn are divided into cities. We use this same hierarchical organization into domains for our simulation experiments.

Figure 2 shows an object O residing in domain Perth that is looked up by a client in domain Pretoria. The lookup request travels from the server in this lowest-level domain to the server in Cape Town (domain South Africa), to the server in Nairobi (domain Africa) until it reaches the object's server in Washington DC (for the top-level domain). From there, the request follows a path of forwarding pointers to Jakarta (domain Australasia), Sydney (domain Australia), and finally Perth where the object now resides.

A better mapping for this situation would have been to place the object's top-level server in New Delhi or even Jakarta. Of course, the appropriateness of the mapping depends on where the object currently resides and where lookup requests come from. Dynamically changing a mapping would perhaps be the best thing to do, but this turns out to be difficult as we discuss below.

Note that the search tree for a given object resulting from the organization of the network into a hierarchy of domains guarantees that lookup requests never travel outside the smallest domain containing both client and object. In other words, if our example client was located in Canberra

instead of Pretoria, the lookup request would go from Canberra to Sydney and from there to Perth. An analogous reasoning holds for updates. In this sense, requirement R3 states that we should additionally minimize the communication when switching from one domain to another (higher-level or lower-level) domain.

3.1 An Efficient Mapping

A location service that is distributed worldwide should preferably have the following property. When a client C issues a request to lookup an object O, the request should travel along a path of location servers that corresponds to the (optimal) network route that any message from C to O would follow. In low-level domains, this routing aspect plays a less prominent role compared to routing between location servers at high-level domains.

Basic Approach

Returning to our example, suppose a location server S_1 in a subcontinent receives a lookup request for an object O that it has no information on. We need to decide what the best server S_0 in the top-level domain is to which S_1 should forward the request. As S_1 has no clue on the whereabouts of O, we can only resort to a good heuristic. In our case, we assume that an object *generally* resides in the vicinity of its **home location**. The home location is assumed to be the place where the object is created. Below we discuss what needs to be done in those cases for which this assumption fails. Instead of using only object identifiers, we make use of an **object handle** that contains an object's OID as well the coordinates of the location where an object was created (i.e., its home location).

In our approach, we let an LS host run one server for each domain in which that host is contained. (For example, this is the case only for hosts 2, 4, 5, and 8 shown in Figure 1.) In other words, if A_{home} is the address where object O is created, then all servers $S_k(A_{home})$ run on the same LS host H_{home}. H_{home} itself is located somewhere in $D_n(A_{home})$, that is, in the lowest-level domain where O was created.

In our example from Figure 2, if object O was created in Jakarta, then its servers for respectively the top-level domain, domain Australasia, domain Indonesia, and domain Jakarta would all run on the same host, which is somewhere in Jakarta. With this mapping, the lookup request initiated in Pretoria would travel from Pretoria to Cape Town (domain South Africa), to Nairobi (domain Africa), to Jakarta (domains World and Australasia), to Sydney (domain Australia), and finally to Perth.

Now consider an arbitrary domain D. Taking a specific distance metric such as the geographical distance, we always select the location server S for O in D that is closest to H_{home}. Again, note that if O is residing somewhere else than in D, requests will still travel a route that exploits locality. We return to choosing a suitable distance metric below.

A simple, but computational expensive implementation of $map(O,D)$, is to take the coordinates of the home location contained in the object handle for object O and compute the distance to each server in D. We then select the server that is closest to O's home location. We can trade time for space by computing distances in advance and storing the selected server in **location-mapping table**, which uniquely associates a location server to O in D. To construct such as

table, we divide the surface of the earth into a large number of small disjoint **elementary areas**. This division is independent from the organization of the network into a hierarchy of domains. A straightforward way to create elementary areas is by means of a grid 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, $0.25° \times 0.25°$ degree areas, this results in $1,036,800$ elementary areas. Location coordinates of a home location are now expressed as the elementary area where an object was created.

Table Implementation

If the number of elementary areas is not too large, we can efficiently implement the mapping table as a 2-dimensional array. In this solution, the (x,y) coordinate representing an elementary area is used as an index. To get an impression of the storage size of the array, assume that each location server is represented by a **server identifier** having a size $m = 18$ bytes. Such an identifier could be formed as the combination of a 16-byte IPv6 address of the LS host that is running the server, and a 2-byte port number for that server. To compress storage, we use indirect indexing. We store the server identifiers in a separate table of size N, where N is the total number of servers. Each indirect index will require $n = \lceil \frac{1}{8} \log_2 N \rceil$ bytes. The mapping table stores the indices to this separate table. If there are E elementary areas, the total required storage is $E \cdot n + N \cdot m$ bytes. With $E = 1,036,800, N = 10,000$, and $m = 18$, a mapping table requires approximately 2.1 MB.

A server for a domain D will have to maintain a mapping table for each of the child domains of D, as well as a table for the parent domain of D. Not all tables are of the same size. However, even if we assume that all tables require 2.1 MB storage, it can be shown that for the example domain partitioning we have used in our experiments, each LS host will have to reserve 140 MB for table storage. We do not consider this a problem. When elementary areas are chosen smaller, we may need to apply table reduction methods. Typically, we can use the same methods as applied in geographical databases such as quadtrees [20]. We are currently investigating the applicability and effect of these methods.

3.2 Mapping Management

Let us consider some of the issues related to maintaining the scheme.

Handling Outdated Object Handles

It may well be that in the course of time an object's home location changes due to a more or less permanent move. Such a migration may make the mapping to location servers for that object inefficient. For example, an object created in Washington DC that has permanently moved to Perth will still have its top-level server in Washington DC leading to communication patterns as shown in Figure 2.

A solution to this problem is to subject the location information contained in an object handle to a lease [8]. Effectively, when the lease expires, the object handle becomes invalid and a client is forced to lookup a fresh handle for the object using its OID. This solution requires a separate globally available service that binds an OID to an object handle. Because it can be expected that such a binding hardly changes, efficiently implementing such a service in a scalable way is not

very difficult. For example, the place where an object was created can host a service that keeps track of the object's current handle. The location where an object was created will need to be encoded in the OID, but will now never change.

Adding and Removing Servers

Having a large collection of LS hosts we can expect that the mapping needs to be changed regularly as hosts, and thus their servers, come and go. As a consequence, location-mapping tables will need to be updated regularly and propagated to the appropriate hosts. When removing or adding a server for a low-level domain, the situation is relatively simple because this will affect only a relatively small number of hosts. However, when adding or removing a top-level server, in principle *all* LS hosts will need to be informed.

We adopt a simple solution. Whenever a server is added or removed, this change is recorded in a globally distributed database that can be accessed at well-known locations. Once every T time units, a host can update its tables by contacting this database. A host immediately contacts this database when it notices a server has been brought down. Likewise, by using version numbers it can also detect that its tables are outdated when contacting another host. With this approach, it will take at most T time units for a new server to become fully operational.

3.3 Requirements revisited

This implementation of $map(O,D)$ fulfills most of the requirements. Requirement $R1$ is obviously met. By using quadtrees, the size and computational complexity of computing $map(O,D)$ is low, in accordance with requirement $R2$. If we assume that the shortest geographic distance corresponds to shortest network distance, we also satisfy $R3$. This is a strong assumption that does not hold in the current Internet (see also [3]). However, any distance metric for which the triangulation inequality holds (i.e., a metric that is Euclidean) suffices. For example, the number of network-level routing hops would do as well. Using a different metric may possibly lead to a different partitioning of the network into a hierarchy of domains, but would otherwise not affect our mapping scheme. In this sense, we are confronted with the same problems as number-based routing in peer-to-peer networks, which also requires taking network proximity into account in order to be efficient [19, 26]. Requirement $R4$, which states that it should be easy to change the mapping tables, requires that we maintain a (possibly replicated) global database from which new tables can be downloaded. This solution is not entirely satisfactory, but suffices for now. Requirement $R5$ is also met, as we demonstrate below.

4 Evaluation

We simulated the behavior of the location service across a worldwide network that connects all cities having at least 100,000 inhabitants. The goal of our experiment is to see to what extent our basic mapping scheme establishes good load balancing while preserving locality. Data from 1986 and 1987 on these cities have been collected in the **World Cities Population Database** (**WCPD**) [18].[1] The database contains records of approximately 2500 cities, each with their

[1] These data are available at `http://www.grid.unep.ch/data/grid/gnv29.html`.

Subcontinent	Contries	Cities	Population
North America	11	274	98029259
South America	13	240	96418651
Europe	29	436	142158983
Africa	32	124	40157801
East Asia	4	493	271493831
South Asia	17	336	130192023
North Asia	2	287	112019100
Australasia	14	109	49271682
Total	122	2299	939741330

Figure 3: The division of the world into subcontinents.

population size, geographical position, and country. WCPD also has records on smaller cities that are capital of a country. We have excluded these cities from our experiment, which were approximately 200 in total. The remaining 2299 cities jointly populated nearly 1 billion people in 1987.

We treat all cities equal in the sense that we assume they generate in proportion to their population, as many requests for objects as other cities. Additionally, we make the following assumptions:

- Each object has only a single contact address (i.e., we do not consider replicated objects).
- A contact address is always stored in a server for a lowest-level domain. Intermediate nodes store only forwarding pointers.
- There are no location caches as described in [2, 10].
- The communication delay between two hosts can be expressed as a linear function of the distance between those hosts.

As we discussed above, the last assumption is not realistic for the current Internet. However, any Euclidean metric will suffice. Taking costs expressed in terms of another metric would lead only to a different partitioning into domains, but would not affect the final conclusions.

4.1 Modeling Issues

We divide the world into four different types of domains: the world, subcontinents, countries, and cities. A subcontinent is a large geographical area covering several countries. In our experiment, we distinguished eight subcontinents, also shown in Figure 2. The subcontinents are North America (including Central America), South America, Europe (including the Mediterranean area), Africa, East Asia, South Asia (including a number of countries from the Middle East), North Asia, and Australasia. This partitioning gives the number of cities and population shown in Figure 3.

Each domain has at least one associated server. For simplicity, we assume there is a single host available in each city. In practice, each such host would be presumably be implemented as a (distributed) cluster of machines that effectively operates as a high-performance multicomputer.

Each object is represented only by the city where it was created. Given an object O and a domain D, we compute the location of the host in D that is responsible for handling requests for O. We compare six different mapping strategies, summarized in Figure 4. Strategy

Mapping	Description
MAP_LOC_AWARE	For the current domain, select the host closest to city where *O* was created.
MAP_RANDOM	Randomly select a host in *D* to handle all requests for *O*.
MAP_LA1&RND	Apply *MAP_LOC_AWARE* for level-0 domain and *MAP_RANDOM* for other domains.
MAP_LA2&RND	Apply *MAP_LOC_AWARE* for level-0 and level-1 domains and *MAP_RANDOM* for other domains.
MAP_LOGICAL	For the current domain, select the host in the center of *D*.
MAP_HOME	Regardless the current domain, select the host in the city where *O* was created.

Figure 4: The six mapping strategies.

MAP_LOC_AWARE is the one described in Section 3. It selects the host closest to where *O* was created. This strategy should be better in terms of using network resources than *MAP_RANDOM*, which randomly selects one of the hosts in *D* to handle all requests for *O*. Strategy *MAP_LA1&RND* combines the previous two by selecting the closest host for the top-level domain, but a random one at other levels. As we shall see, this strategy leads to a better load balancing compared to *MAP_LOC_AWARE*. Strategy *MAP_LA2&RND* applies *MAP_LOC_AWARE* both for the top-level domain and subcontinents, but randomly selects hosts at all other levels.

For comparison, we also consider constructing a single tree that is to handle the entire collection of objects. In strategy *MAP_LOGICAL*, all objects are associated to the same server, namely the one closest to the center of a domain. We determine this center by computing, for each city in a domain, the aggregated distance to all other cities in that domain. The city with the smallest aggregated distance is chosen as the center.

A simple, effective, and widely applied strategy for locating mobile objects is to introduce a single server for each object and let that server keep track of an object's current location. These home-based approaches are used in mobile IP [15], but also wireless telephony [14]. Our discussion on hierarchical solutions makes sense only if these solutions show to be better than home-based approaches. For this reason, we also consider the strategy *MAP_HOME*, by which all requests for an object are always forwarded to a server running on a host in the city where the object was created.

Mobility and Lookup Patterns

To simulate mobility and lookup patterns we adopt the following model. Let A_{home} be the address of the location where object *O* was created. The level of a domain is indicated by a subscript *k*. Before simulating a move or lookup operation on object *O*, *O* is placed in a city randomly chosen from domain $D_k(A_{home})$. Domain $D_k(A_{home})$ is selected with probability $p_{init,k}$. For example, an object created in Perth will initially be placed somewhere in domain Australasia with probability $p_{init,1}$.

For any object *O*, we assume there is a probability $p_{move,k}$ that *O* will move to a city randomly chosen from $D_k(addr(O))$. Likewise, there is a probability $p_{lookup,k}$ that a lookup request for *O* comes from a city chosen in domain $D_k(addr(O))$. This city from where the lookup comes from is chosen according to a uniform distribution taking the population size as a weight factor (i.e., larger cities are chosen more often than smaller cities). In our simulations, we have chosen the ratio between move and lookup operations for any object *O* equal to 0.2.

Model	$p_{init,k}$				$p_{move,k}$				$p_{lookup,k}$			
k:	3	2	1	0	3	2	1	0	3	2	1	0
UUU	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
ULL	0.25	0.25	0.25	0.25	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10
ULU	0.25	0.25	0.25	0.25	0.50	0.25	0.15	0.10	0.25	0.25	0.25	0.25
LUU	0.50	0.25	0.15	0.10	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
LLL	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10
LLU	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10	0.25	0.25	0.25	0.25

Figure 5: The six different models used for simulations.

Models for Mobility

Figure 5 shows the six different models for mobility that we used for our simulations. Each model has a 3-character mnemonic, each character denoting a uniformly distributed (U) or localized (L) pattern for initial placement, migration, and lookups, respectively. In the first three models, we assume that the initial placement of an object is uniformly distributed across all levels. For model *UUU*, we also assume that after the initial placement the probability that a migration will take place within, respectively, the same city, country, subcontinent, or anywhere, is the same. Likewise, each domain $D_k(addr(O))$ has the same probability for generating a lookup request.

In model *ULL*, we assume that an object generally makes only local movements, and likewise, that most lookup request come from the same city where the object is now located. Model *ULU* reflects that objects generally move locally, but gives an equal probability that a lookup comes from the same city, country, subcontinent, or from anywhere.

The last three models are analogous to the first three, except that we make the assumption that the initial placement is generally in the same city as where the object is created.

Simulation

For each run, we generate 100 million requests. To select an object, we choose a city following a weighted uniform distribution that takes the population size of each city into account as explained above. Simulating a request starts with choosing a domain $D_k(A_{home})$ with probability $p_{init,k}$ from which we randomly select a city for the initial placement of O.

For a move request, we then select the domain $D_k(addr(O))$ with probability $p_{move,k}$ and choose a source and destination city in this domain. The effects of the migration are simulated by registering an update at all relevant servers. As we explained, migration involves handling an insert request for an address at the destination, and a delete request at the source. Each request travels from a server in a lowest-level domain (located in a city) to the object's server in $D_k(addr(O))$. In our simulation, we add the distances that the insert and delete request travel, respectively.

For a lookup request, we pretend the object has moved after its initial placement by selecting a domain $D_{move,k}(A_{home})$ with probability $p_{move,k}$ and choosing a city from that domain as the object's current location. We then select a domain $D_{lookup,k}(addr(O))$ with probability $p_{lookup,k}$ and choose a city in $D_{lookup,k}(addr(O))$ from where a lookup request is generated. This city is chosen by taking the population sizes of all cities in $D_{lookup,k}(addr(O))$ into account. The effect of the request is measured by registering that a lookup operation is processed at all relevant servers, as well as measuring the distance the request needs to travel before reaching the location server

at the object's current location.

4.2 Results

As we mentioned, the goal of our simulations is to evaluate to what extent load balancing is achieved while preserving locality for lookup and update operations. We first consider the load distribution across the hosts. We counted the number of lookup and move operations that each server (and thus its host) needed to perform and compared that to the total number of operations that were carried out. Figure 6 shows the accumulated number of hosts that take care of processing an increasing fraction of operations. Hosts have first been sorted by their load; a higher load leading to a higher ranking. If we had perfect load balancing, we would see a straight line from coordinate (0, 0) to (2299, 1). However, this is not the case.

For example, we see that with strategy *MAP_HOME* only 263 of the 2299 servers are responsible for handling 50% of all operations. This is not surprising considering our assumption that each city has only a single host, while at the same time we assume that larger cities generate more requests.

As it turns out, *MAP_LOGICAL* shows bad load balancing for all models. Of course, this was to be expected: the single root server and the few subcontinent servers will see most of the requests. When taking a look at the load distribution for *MAP_LOC_AWARE*, we see that it tends to follow a similar distribution as that for *MAP_HOME*. However, it should be noted that servers in the *MAP_HOME* approach generally need to process only 25% of the operations compared to the other strategies. This difference is due to the fact that we have a tree of height four.

Strategy *MAP_RANDOM* comes close to a perfect load distribution. We have not shown the load distribution for *MAP_LOC1&RND* as it almost identical to that of *MAP_RANDOM*. However, note that the distribution for *MAP_LOC2&RND* also comes close to that of *MAP_RANDOM*. As we discuss below, *MAP_LOC2&RND* also exhibits locality, making it a good overall strategy.

To see to what extent locality was preserved, we also measured the network traffic that was generated. In particular, we measured for each request the geographical distance that it traveled before reaching a server where a contact address was found. In the case of migrations, we measured the distance needed to complete the combination of an insert request and a delete request. The aggregated results are shown in Figure 7. Except for models *UUU* and *LUU*, which reflected almost no locality in the lookup and migration patterns for objects, strategy *MAP_LOGICAL* gives the best results, closely followed by *MAP_LOC_AWARE* and *MAP_LOC2&RND*. The home-based approach gives the best result when there is hardly any locality.

We also measured locality by computing the fraction of requests against the *maximum distance* that needed to be traveled. We have left out *MAP_RANDOM* and *MAP_LOC1&RND*, but concentrate on the more promising strategy *MAP_LOC2&RND*. The distances travelled by lookup operations are shown in Figure 8, those for migrations are shown in Figure 9. First, consider model *UUU*. All strategies show roughly the same behavior when it comes to exploiting locality. Nevertheless, in this model, *MAP_HOME* is better because *all* requests need to travel at most 20,000 kilometers (i.e., half of the earth's circumference) whereas in the other strategies approximately 10% of the requests travel a longer distance. However, when we consider model

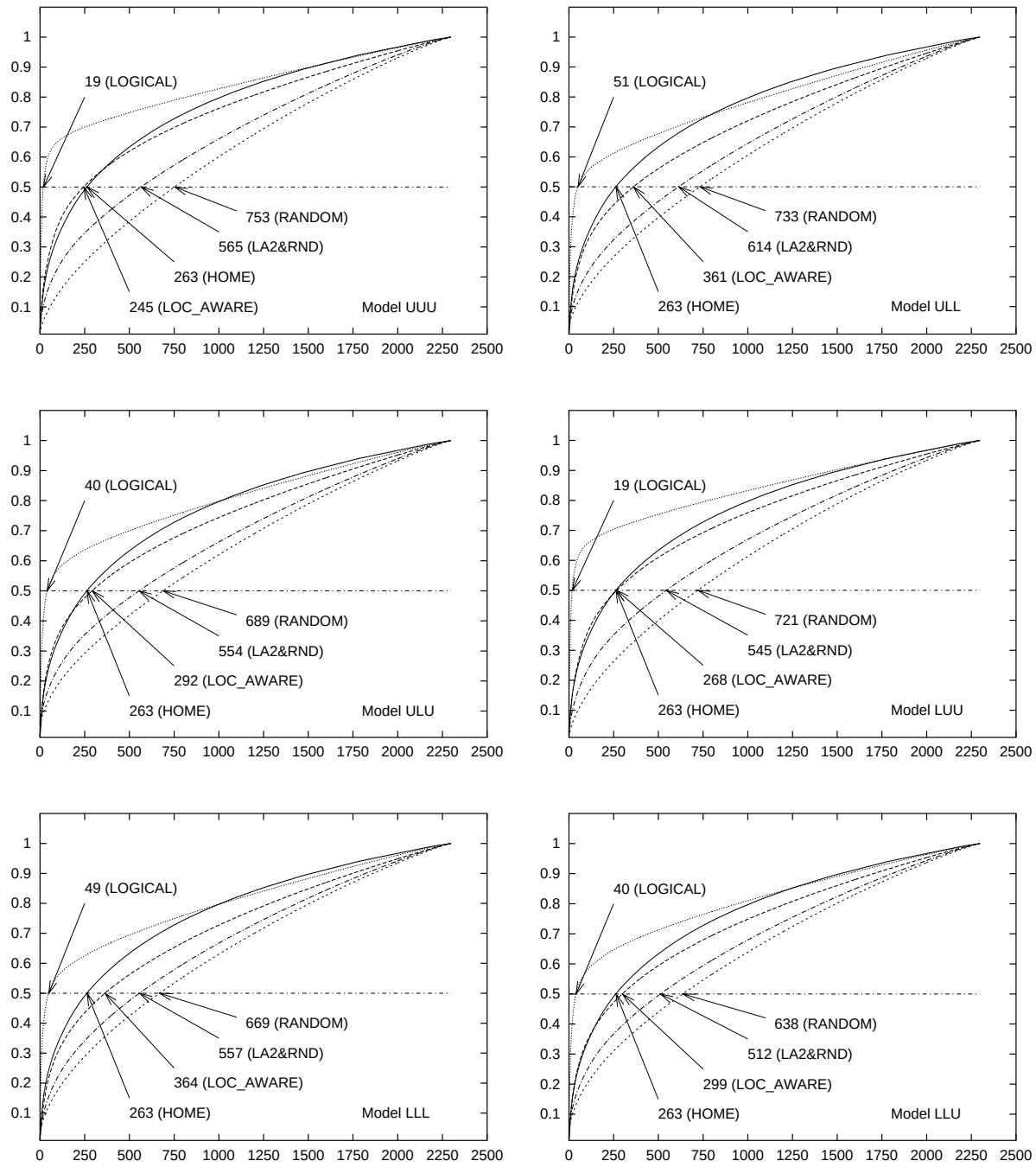
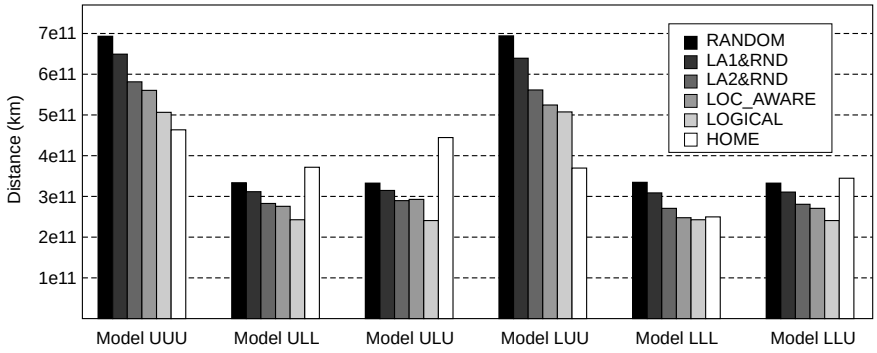


Figure 6: Load distribution for the six models. The *x*-axis shows the accumulated number of hosts after sorting hosts by their load. The *y*-axis shows the fraction of operations that take place.



Model	Mapping strategy					
	RANDOM	LA1&RND	LA2&RND	LOC_AWARE	LOGICAL	HOME
UUU	6.94e11	6.50e11	5.82e11	5.61e11	5.07e11	4.64e11
ULL	3.34e11	3.12e11	2.83e11	2.76e11	2.43e11	3.72e11
ULU	3.33e11	3.15e11	2.90e11	2.93e11	2.41e11	4.45e11
LUU	6.95e11	6.40e11	5.62e11	5.25e11	5.08e11	3.70e11
LLL	3.35e11	3.09e11	2.71e11	2.48e11	2.43e11	2.50e11
LLU	3.33e11	3.11e11	2.81e11	2.71e11	2.42e11	3.45e11

Figure 7: Total geographical distance (in kilometers) traveled by requests before completion.

ULU, for example, we see that the hierarchical organization is more successful in exploiting locality. In this model, when we consider the maximum distance traveled by 75% of all lookup requests, the home-based approach gives 7228 km, whereas the hierarchical approach gives a maximum of 3359 km.

Figure 9 shows the distance that the requests involved in a move operation travel. It is analogous to Figure 8. Note that when considering only migrations, models *ULL* and *ULU*, as well as models *LLL* and *LLU* will show the same results.

We conclude that our mapping strategy establishes load balancing while preserving locality.

5 Conclusions

The argument that hierarchical location services introduce a scalability problem for higher-level nodes is not true. It is possible to design a scheme by which location information is distributed in such a way that the load between hosts is well balanced and largely independent of lookup and mobility patterns. Moreover, our study shows that good load balancing can be combined with exploiting locality, a property that home-based approaches generally do not have.

There are two major drawbacks of hierarchical solutions in comparison to home-based solutions. First, to exploit locality, we are forced to generally forward requests between several location servers instead of just one. Not only may this lead to additional delays at the requester's side, it also increases the load on servers. On the other hand, exploiting locality reduces the use of network resources. The second drawback is the management of mapping tables. Our global solution is simple and effective, but a local solution that can be computed in isolation would be preferable. Finding such a solution is subject to current research.

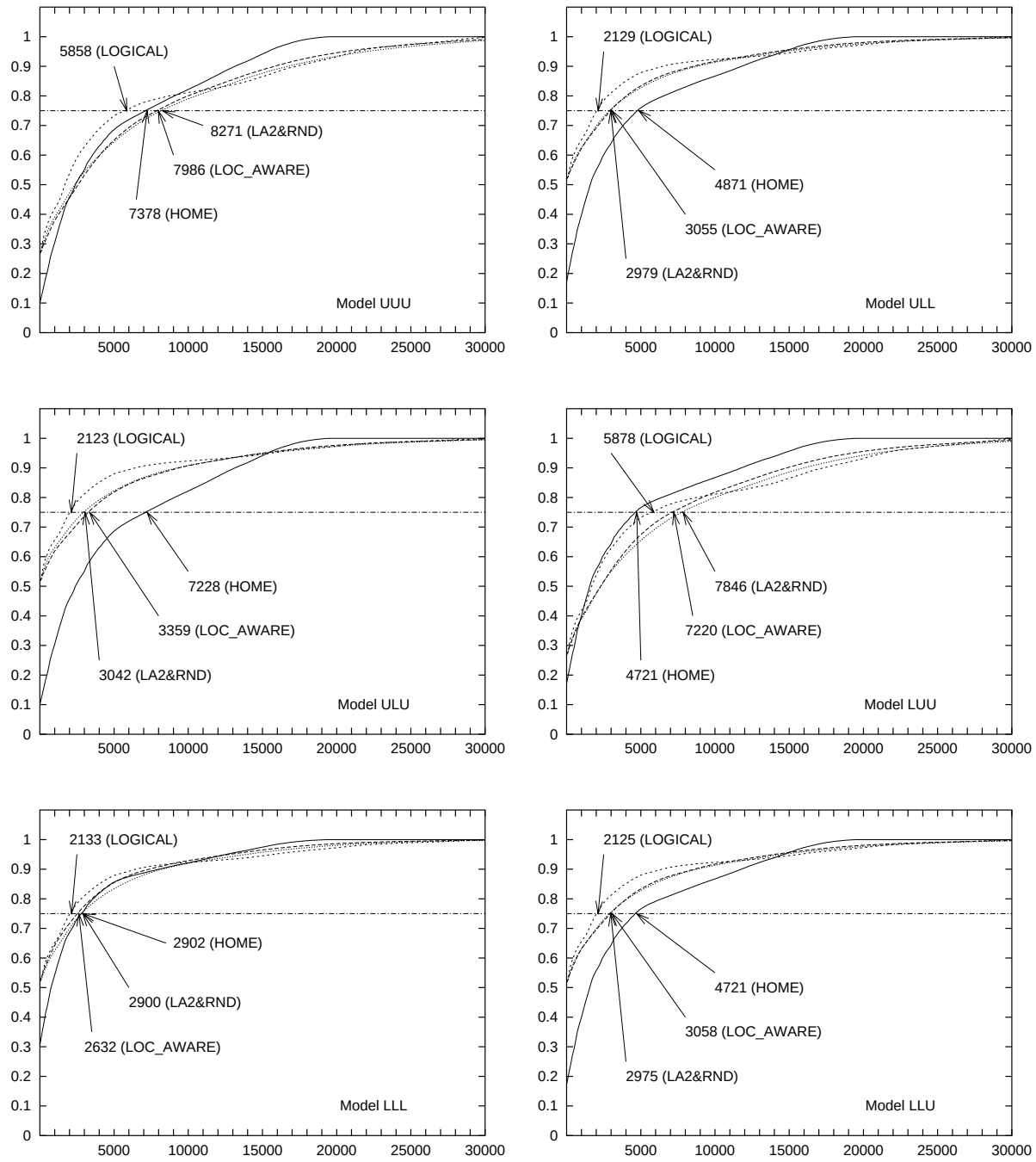


Figure 8: The distance that lookup requests travel. The *x*-axis shows the maximum distance that a request travels. The *y*-axis shows the fraction of lookup requests.

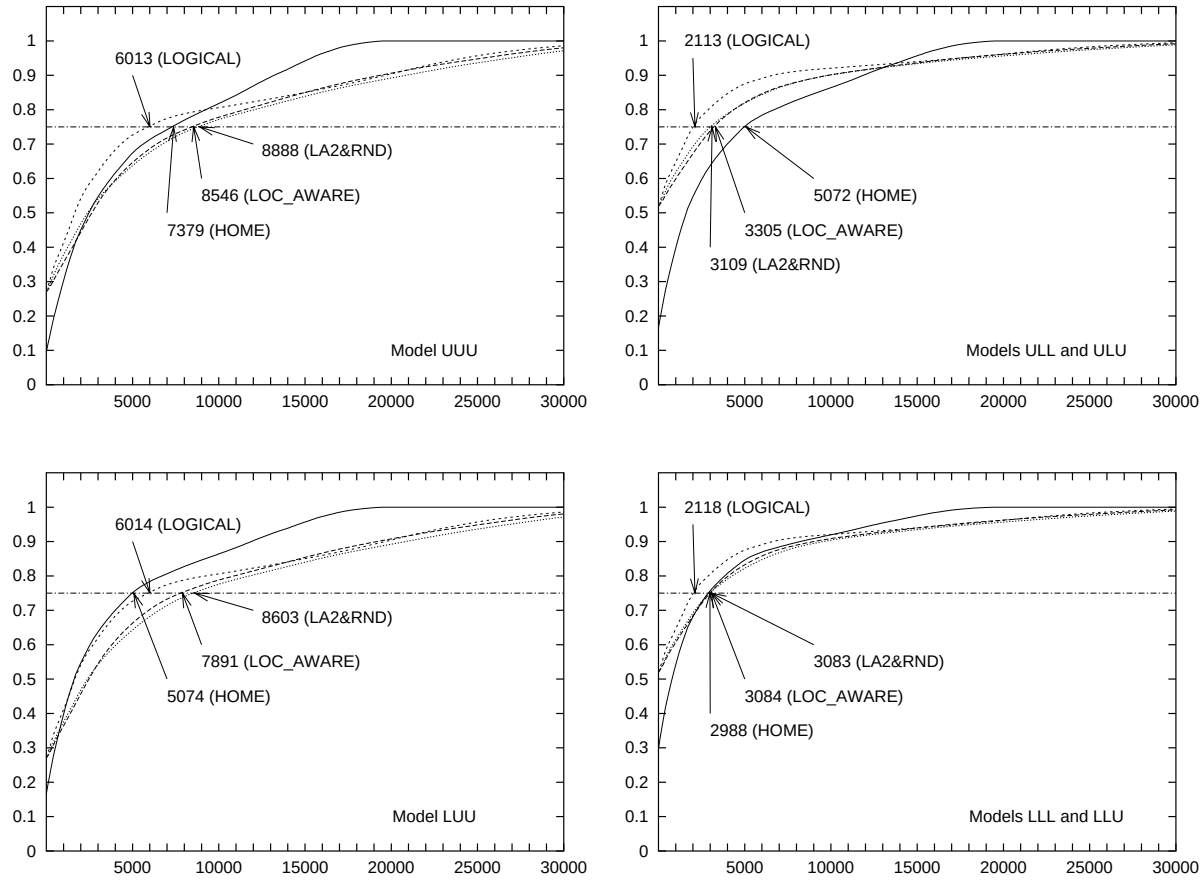


Figure 9: The distance that migration requests travel. The *x*-axis shows the maximum distance that a request travels. The *y*-axis shows the fraction of migration requests.

References

[1] B. Awerbuch and D. Peleg. "Online Tracking of Mobile Users." *J. ACM*, 42(5):1021–1058, Sept. 1995.

[2] A. Baggio, G. Ballintijn, M. van Steen, and A. Tanenbaum. "Efficient Tracking of Mobile Objects in Globe." *Comp. J.*, 44(5):340–353, 2001.

[3] G. Ballintijn, M. van Steen, and A. Tanenbaum. "Characterizing Internet Performance to Support Wide-area Application Development." *Oper. Syst. Rev.*, 34(4):41–47, Oct. 2000.

[4] Y. Bejerano and I. Cidon. "Efficient Mobility Management Schemes for Personal Communication Systems." Technical Report CC-PUB-271, Center for Communication and Information Technologies, Technion Haifa, Israel, Mar. 1999.

[5] L. Bernardo and P. Pinto. "A Scalable Location Service Supporting Overload Situations." In *Proc. Third Int'l Workshop on Artificial Intelligence in Distributed Information Networking*, pp. 51–56, Orlando, FL, July 1999. AAAI.

[6] A. Black and Y. Artsy. "Implementing Location Independent Invocation." *IEEE Trans. Par. Distr. Syst.*, 1(1):107–119, Jan. 1990.

[7] R. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. Ph.D. thesis, University of Washington, Seattle, 1985.

[8] C. Gray and D. Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." In *Proc. 12th Symp. Operating System Principles*, pp. 202–210, Litchfield Park, AZ, Dec. 1989. ACM.

[9] Y. Hu, D. Rodney, and P. Druschel. "Design and Scalability of NLS, a Scalable Naming and Location Service." In *Proc. INFOCOMM*, New York, NY, June 2002. IEEE.

[10] R. Jain. "Reducing Traffic Impacts of PCS using Hierarchical User Location Databases." In *Proc. Int'l Conf. Communications*, Dallas, TX, June 1996. IEEE.

[11] E. Jul, H. Levy, N. Hutchinson, and A. Black. "Fine-Grained Mobility in the Emerald System." *ACM Trans. Comp. Syst.*, 6(1):109–133, Feb. 1988.

[12] P. Krishna, N. Vaidya, and D. Pradhan. "Static and Dynamic Location Management in Distributed Mobile Environments." *Comp. Comm.*, 19(4), Mar. 1996.

[13] C. Leiserson. "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing." *IEEE Trans. Comp.*, 34(10):892–901, Oct. 1985.

[14] S. Mohan and R. Jain. "Two User Location Strategies for Personal Communication Services." *IEEE Pers. Commun.*, 1(1):42–50, Jan. 1994.

[15] C. Perkins. *Mobile IP: Design Principles and Practice*. Addison-Wesley, Reading, MA, 1997.

[16] E. Pitoura and I. Fudos. "An Efficient Hierarchical Scheme for Locating Highly Mobile Users." In *Proc. Sixth Int'l Conf. on Information and Knowledge Management*. ACM, Nov. 1998.

[17] E. Pitoura and G. Samaras. "Locating Objects in Mobile Computing." *IEEE Trans. Know. Data Eng.*, 13(4):571–592, July 2001.

[18] D. Rhind. "Cartographically-related research at Birkbeck College 1987-91." *The Cartographic Journal*, 28:63–66, June 1991.

[19] A. Rowstron and P. Druschel. "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems." In R. Guerraoui, (ed.), *Proc. Middleware 2001*, volume 2218 of *Lect. Notes Comp. Sc.*, pp. 329–350, Berlin, Nov. 2001. Springer-Verlag.

[20] H. Samet. "The Quadtree and Related Hierarchical Data Structures." *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[21] M. Shapiro, P. Dickman, and D. Plainfosse. "SSP Chains: Robust, Distributed References Sup-

porting Acyclic Garbage Collection." Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." In *Proc. SIGCOMM*, San Diego, CA, Aug. 2001. ACM.

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

[24] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, 36(1):104–109, Jan. 1998.

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

[26] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing." Technical Report CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr. 2001.