Achieving Scalability in Hierarchical Location Services

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

Abstract

Services for locating mobile objects are often organized as a distributed search tree. The advantage of this approach is that the service can scale as a system grows in size and number of objects. However, a potential problem is that high-level nodes may become a bottleneck affecting the scalability of the service. A traditional solution is to also distribute the location information managed by a single node across multiple machines. 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.*

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 [5]. 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 [8], 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. More information can be found in the extended version of this paper [7].

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.

2.1 General Model

A (mobile) entity is assumed to be represented by a single object with 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 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

0730-3157/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

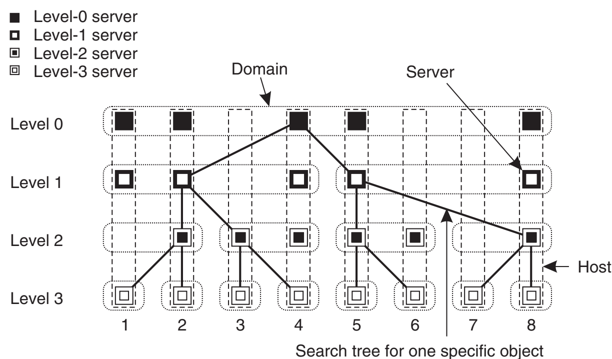

Figure 1. An example of organizing the collection of LS hosts into a hierarchical organization of domains, with each domain having at least one location server.

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). We thus guarantee deterministic behavior of lookup and update algorithms.

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. 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 shown in Figure 1. Note that the collection of root nodes of these trees may be completely distributed across the servers in the top-level domain.

2.2 Operations

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.

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 [5]. 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

0730-3157/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

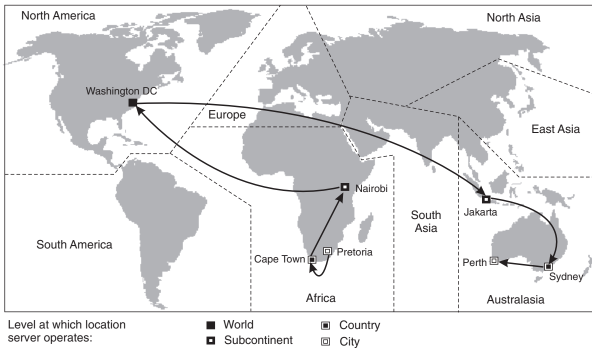

Figure 2. A wrong mapping introduces more communication.

and immediately redirect lookup requests to D, thus avoiding that high-level nodes need to process the request.

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 $D_k(B)$ [4]. 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. 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. 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 [1] and resembles the number-based routing as applied in peer-to-peer networks [6]. It works fine for systems in which locality is not really an issue, 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 requirement that 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. 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 in [7].

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.

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. As we discuss in [7], this is not a hard assumption and with some effort we can adjust our mapping for objects that permanently move to another home location. 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.

IEEE COMPUTER SOCIETY

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.

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**).[1] The database contains records of approximately 2500 cities, each with their population size, geographical position, and country. For our study, we concentrated only on the 2299 cities with 100,000 or more inhabitants.

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. In addition, we make the important assumption that the communication delay between two hosts can be expressed as a linear function of the distance between those hosts. This assumption is not realistic for the current Internet. However, 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.

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 presumably be implemented as a (distributed) cluster of machines that effectively operates as a high-performance multicomputer.

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

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_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 3. The five mapping strategies.

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 five different mapping strategies, summarized in Figure 3. Strategy *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_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 [3], but also wireless telephony [2]. 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

IEEE COMPUTER SOCIETY

$D_k(addr(O))$. 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.

Models for mobility. Using different values for these three probabilities, we experimented with six different models for mobility. 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. The models we examined where *UUU*, *ULL*, *ULU*, *LUU*, *LLL*, and *LLU*. 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.

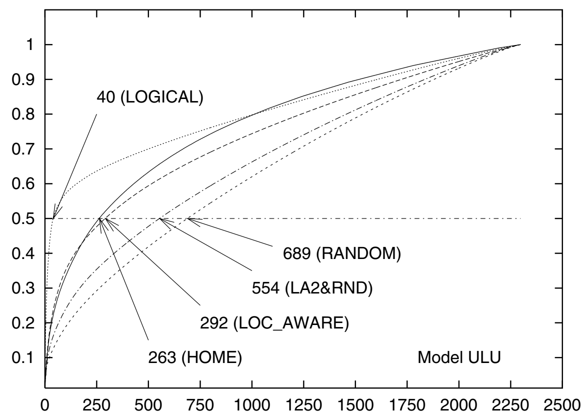

Figure 4. Load distribution for model *ULU*. The *x*-axis shows the accumulated number of hosts after sorting hosts by their load. The *y*-axis shows the fraction of operations that take place.

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 4 shows the accumulated number of hosts that take care of processing an increasing fraction of operations. We show only the results for model *ULU*; the other models show similar results as can be found in [7]. 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 we also assume that larger cities generate more requests.

IEEE COMPUTER SOCIETY

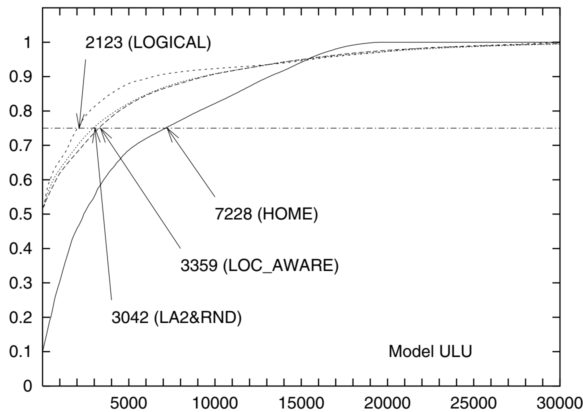

Figure 5. The distance that lookup requests travel for *ULU*. The *x*-axis shows the maximum distance that a request travels. The *y*-axis shows the fraction of lookup requests.

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. 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. Figure 5 shows the results for lookup operations in model *ULU* (again, the results for other distributions are comparable [7]). 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. We conclude that our mapping strategy establishes load balancing while preserving locality.

5 Conclusions

In order for systems to expand across wide-area networks and to scale in terms of the number of objects they support, services should sustain and scale along with the system. Location services for tracking and looking up objects are important in any distributed system. To meet scalability requirements, such services are often constructed as a distributed search tree that spans the network underlying the distributed system.

However, the argument that these hierarchical location services introduce a scalability problem for higher-level nodes is not true. We have shown that 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. In the extended version of this paper [7], we also show that the load is also 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.

References

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

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

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

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

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

[6] C. Plaxton, R. Rajaraman, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. *Theory of Computing Systems*, 32:241–280, 1999.

[7] M. van Steen and G. Ballintijn. Achieving Scalability in Hierarchical Location Services. Technical Report IR-491, Vrije Universiteit, Department of Mathematics and Computer Science, Nov. 2001.

[8] 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.