Locating Objects in Wide-Area Systems

Maarten van Steen (contact)

Vrije Universiteit, Department of Mathematics & Computer Science
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands

tel: +31 (0)20 444 7784, fax: +31 (0)20 444 7653
e-mail: steen@cs.vu.nl

Franz J. Hauck

University of Erlangen–Nürnberg, IMMD4
Martensstr. 1, D–91058, Erlangen, Germany
e-mail: hauck@informatik.uni-erlangen.de

Philip Homburg
Andrew S. Tanenbaum

Vrije Universiteit, Department of Mathematics & Computer Science
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
e-mail: {philip,ast}@cs.vu.nl

Locating Objects in Wide-Area Systems

Maarten van Steen, Franz J. Hauck,
Philip Homburg, Andrew S. Tanenbaum

Abstract

Locating mobile objects in a worldwide system requires a scalable location service. An object can be a telephone or a notebook computer, but also a software or data object, such as a file or an electronic document. Our service strictly separates an object's name from the addresses where it can be contacted. This is done by introducing a location-independent object handle. An object's name is bound to its unique object handle, which, in turn, is mapped to the addresses where the object can be contacted. To locate an object, we need only its object handle. We present a scalable location service based on a worldwide distributed search tree that adapts dynamically to an object's migration pattern to optimize lookups and updates.

1 Introduction

In the near future we can expect hundreds of millions of users to have access to a global Information Superhighway. A large part of that information network will be mobile: telephones, faxes, notebook computers, personal assistants, etc. But we can also expect software and data to be mobile. For example, a Web page may move as its owner changes computers; likewise, a shared electronic document may travel between its users. Another example is a mobile agent that moves through the network in search of specific resources for its owner. Components in a network capable of changing locations, and which may be implemented in software, hardware, or a combination thereof, are collectively referred to as **mobile objects**.

Supporting mobile objects means that a client should be able to contact an object even if he does not know its current location. Moreover, locating the object should be completely hidden from the client. For example, in Personal Communications Systems (PCS), a user should only have to dial a telephone number to contact the callee. It should not be necessary to know the callee's present location or how the callee is tracked. But mobile objects also need to contact other (possibly nonmobile) objects. When a mobile object moves to a new location, the object will have to find out which facilities it can use there. For example, a mobile computer may need to use the local printer. Likewise, it may want to contact the local Web server instead of having to use the server at its home location.

Being able to contact objects, whether they are mobile or not, is traditionally supported by a **naming service** which maintains a **binding** between an object's name and one or more addresses where the object can be contacted. As an analogy, a naming service is like a telephone book; a binding corresponds to one of its entries. With mobile objects, names should always be resolved to a current address. To illustrate, a name such as *pcs://dept.univ.edu/Mary* may be dynamically bound to the network address of Mary's mobile computer. No matter where in the world that name is used, it should always be resolved to her computer's *current* address, which changes as she moves. In addition, applications on her mobile computer may use the name *local://usr/addr/lpr* for the local printer. In this case, as Mary travels around the world, the printer's name on her computer needs to be dynamically rebound to the address of the nearest printer server. Thus, unlike the world of cellular telephony with their fixed bindings of device to telephone number, in the computer world, the addresses used to reach objects change as an object moves, and the mapping of names to addresses must also change.

Changing the address of an object affects the name–to–address binding. If such changes hardly ever occur, then constructing a worldwide scalable naming service is feasible, as demonstrated by the Internet's Domain Name System [8] and the X.500 Directory Service [10]. However, if bindings change frequently, as in the case of mobile objects, we have a much more difficult problem.

In this paper, we focus on a wide-area naming service that provides flexible and easily adaptable name–to–address bindings. The service is currently being developed as part of Globe, an object-based worldwide distributed system aimed to support a billion users each having thousands of objects [3]. The paper is organized as follows. In Section 2 we explain and motivate the basic architecture of the Globe naming service. The main goal of this paper is to explain how objects are located, which is described in Section 3. Related work is discussed in Section 4. We give our conclusions in Section 5.

2 Binding Names to Addresses

To discuss name–to–address binding in wide-area systems, we assume that all (hardware and software) objects have symbolic ASCII names, such as *pcs://dept.univ.edu/Mary*. Also, each object is assumed to have one or more addresses where a client can contact it. By way of analogy, an owner of a cellular telephone is also assumed to have a name which can be registered in a telephone directory. The owner's telephone number corresponds to the address where he can be reached. Unlike cellular telephones, computer objects often have two or more addresses. For example, a replicated file will be known to its users under one name, which is mapped to several addresses, one for each copy. A user asking for the file generally does not care which copy is selected.

To make name resolution efficient for wide-area systems, names often contain location information. For example, the Uniform Resource Locator (URL) *ftp://ds.internic.net/nic/rfc/rfc1737.txt* is the name of a Web page containing the text of RFC 1737. The name reflects where the page is stored (*ds.internic.net*), allowing part of the name resolution process to take place at that location. Similarly, telephone numbers also contain location information: a worldwide number like *+31 20 444 7784* gives the country (31 is The Netherlands), the city (20 is Amsterdam), and a specific telephone exchange (the 444 exchange of the Vrije Universiteit in Amsterdam–Buitenveldert).

However, using location information in names can make it difficult to handle migration. If an object moves, we may have to change its name, or otherwise make that name become a forwarding reference. In wide-area systems, the latter can lead to long chains of references, which are inefficient and susceptible to network failures. What is needed is a naming facility that hides all aspects of an object's location. Users should not be concerned where an object is located or whether it can move.

These requirements can be met by introducing a two-level naming hierarchy as shown in Figure 1. The first level deals with hierarchically-organized, user-defined name spaces. These name spaces are handled by what we call a distributed **object naming service**. A name is bound to an **object handle**, which is a globally unique and location-independent object identifier.

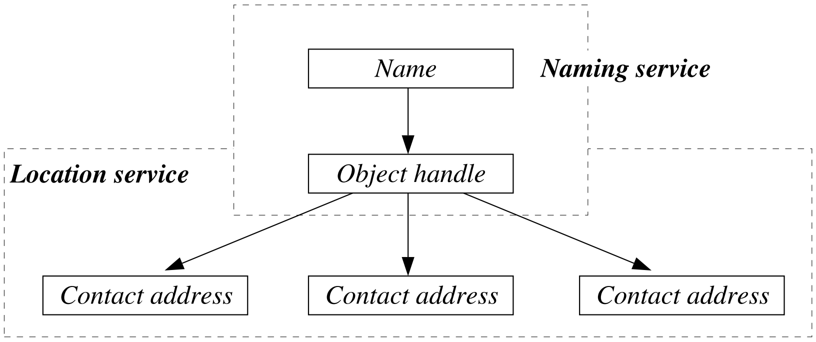

Figure 1: A two-level naming hierarchy that allows an object's name and contact addresses to be independently changed.

The second level deals with mapping each object handle to a set of contact addresses, and is handled by a distributed **object location service**. In contrast to traditional naming services, a location service is designed to support frequent updates and lookups of contact addresses such as needed for mobile objects. It is not concerned with naming.

As an illustration, consider an officeless company whose employees are located across the country, normally working at home, or visiting customers. Using our approach, we assign a lifetime and location-independent telephone number to the company. This number corresponds to an object handle. A naming or directory service would maintain a mapping between the company's name and it's lifetime telephone number. The telephone number is used by a location service to redirect incoming calls to, for example, the nearest employee currently working. An employee's own telephone number is registered at the location service when he or she starts work, and is unregistered again when he or she finishes. An employee's telephone number corresponds to a contact address. Note that how the company is named, and in which directories its name is registered, is no longer important. Naming has been fully separated from how and where we contact the company.

In distributed systems, this way of locating a service is also known as **anycasting**: a client requires a particular service, but is really not interested which server will handle the request. Using our approach, the service is assigned a unique object handle, and each server registers its network address under that object handle. A client has the service's name resolved to the object handle, which is then subsequently resolved to the address of any server that can handle the request

An object handle is designed specifically for the location service. It contains a globally unique *service-independent object identifier* which is very similar to a UUID in DCE [11]. Additionally, an object handle may contain information that can be used to assist in locating the object. For example, it may be known that an object will move only within a certain region. Instead of conducting a global search for such an object, it is more efficient to start the search process in that specific region. Therefore, it makes sense to encode this information into the object handle.

Requirements for a location service

Clearly, our two-level approach makes sense only if we can indeed provide a scalable and efficient naming and location service. The feasibility of developing scalable naming services has been demonstrated by systems such as DNS. This is not yet the case for location services, for which the following requirements will have to be met:

Scalability. The service should allow clients and objects to be located anywhere in the world, and be able to support a huge number of objects. We anticipate that eventually, one billion users with 1000 objects each will be registered with the location service, adding up to 10^{12} objects.

Locality. Assuming that the cost for looking up an address generally increases with the length of the route to that address, we require that the location service exploits locality. This means, for example, that if an object has its address near to the client, finding the object should be fairly cheap.

Stability. Objects may differ with respect to their migration patterns. For example, a Web page may possibly move through the entire network in a seemingly random way, whereas a mobile computer may possibly move only within a city. An object is said to be **stable** with respect to a region, if its addresses most often are in that region. If an object is stable with respect to a region *R*, we require that searching or updating one of its addresses in *R* is cheaper than when the object is not stable with respect to *R*.

Fault tolerance. The location service should be resilient to node and link failures and should continue to operate in the presence of network partitions. The service should, at the least, degrade gracefully in terms of performance and functionality.

Location services are not new and have shown to be relatively easy to implement in local distributed systems. However, they become much more complicated when scalability is taken into account as we discuss next.

3 Tracking Distributed Objects in Globe

In this section we discuss the architecture of Globe's scalable location service. We shall provide only an outline of the architecture, further information can be found in [13].

Basic operations

In our model for tracking objects, we assume a hierarchical decomposition of a (worldwide) network into regions. This decomposition is relevant to only the location service. With each region we associate a **directory node**, capable of storing addresses that lie within that region. This leads to a logical tree-based organization as shown in Figure 2. Addresses are assumed to be location dependent: the region in which an address lies is encoded in the address itself.

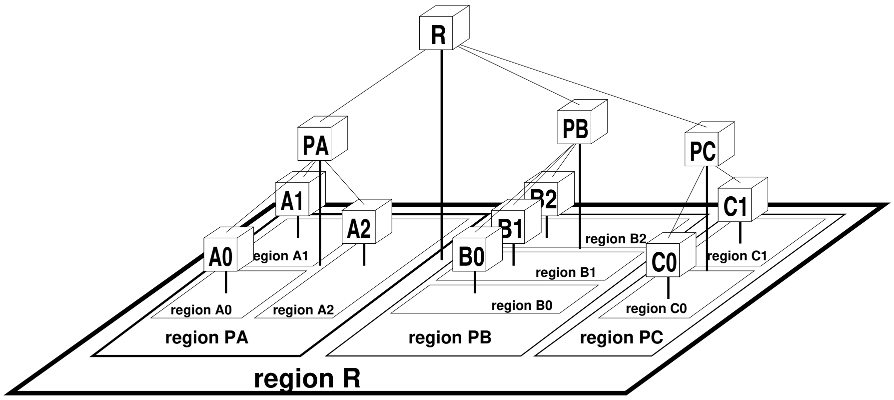

Figure 2: The logical organization of the location service as a virtual search tree.

The location service normally stores new addresses at the leaf node representing the region in which the address lies. For each new object, it constructs a path of forwarding pointers from the root to each leaf node where an address is stored. Addresses and forwarding pointers are stored in **contact records**. An implication of this design is that in the worst case, it is always possible to locate every object by following the chain of pointers from the root node. In practice, we can do much better than this, as described later.

In principle, a request for insertion of a previously unregistered object begins at a leaf node and is propagated up the tree to the root. Then, a path of forwarding pointers is established from the root to the leaf node where the insertion takes place. A contact record containing a forwarding pointer is created at each intermediate node. The address itself is finally stored only in the leaf node. When a part of the path already exists, for example, when inserting a second address in a different region, only the missing pointers are established. This is shown in Figure 3. In the case that there is already a contact record for the object at the leaf node, the new address is simply added to that record.

Deleting a contact address is straightforward and is done as follows. First, the address is found through a search path up the tree, starting at the leaf node representing the region in which the address lies (note that this information is encoded in the address). Once the contact record in which the address is stored, has been found, it is removed from that record. If a contact record no longer contains contact addresses or forwarding pointers, it is deleted. The parent directory node is informed that it should delete its forwarding pointer to that record, possibly leading to the (recursive) deletion of the object's contact record at the parent node.

Looking up a contact address is done as follows. A client process passes an object handle to

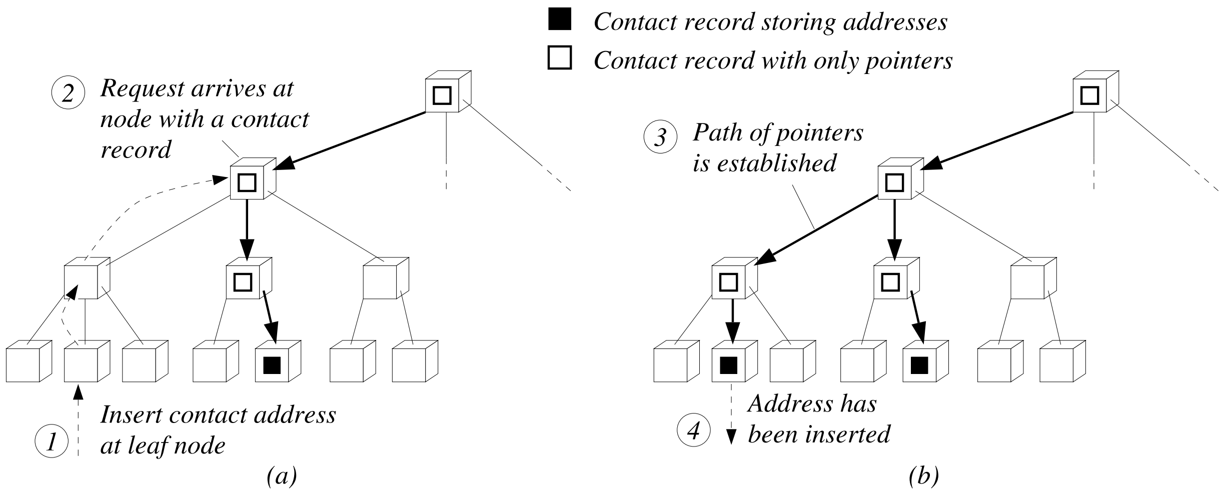

Figure 3: Inserting a contact address when the object is already known. Only the missing pointers are established.

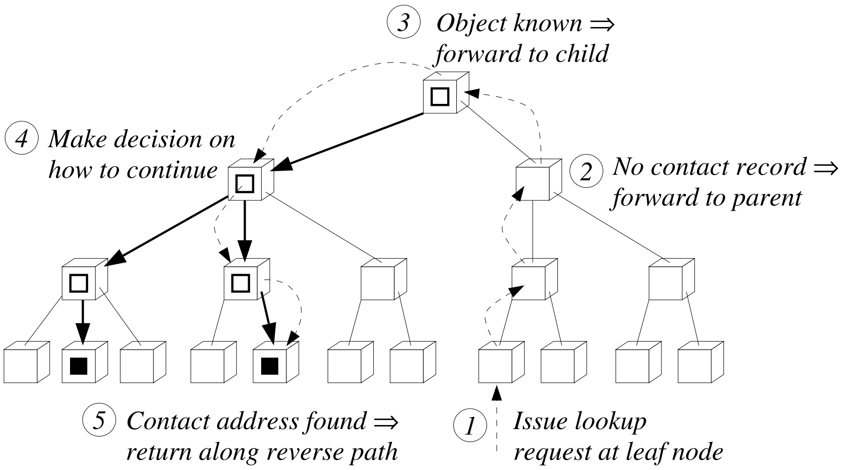

Figure 4: The default approach for looking up a contact address.

the leaf node of the region where that process resides. (We require that there is exactly one such leaf node.) As shown in Figure 4, a search path is established starting at the client's leaf node, and going upwards to the first directory node where the object is already known. In the worst case, this means propagating the request up to the root. The path then continues downwards to a leaf node, whose addresses are then returned to the requester.

Concurrent update and lookup operations are allowed, although no ordering is guaranteed when requests are issued at different nodes. In particular, we have the following consistency rule:

> **Request Consistency**: Update requests issued at the same leaf node are completed in the order they were issued. Update operations issued at different leaf nodes are completed in an arbitrary order.

Dynamic optimizations

The location service has full control over the placement of addresses in contact records. Consequently, if we can place addresses in stable locations, we can make effective use of pointer caches during lookup operations. By default, an object's address is stored in its contact record at the leaf node where it was initially inserted. Now, consider some region R as shown in Figure 5, and assume that an object O is changing its addresses regularly between the subregions S_1, S_2, and S_3. For simplicity, assume that there is always at least one address somewhere in R, so that there will always be a nonempty contact record for O at directory node $dir(R)$.

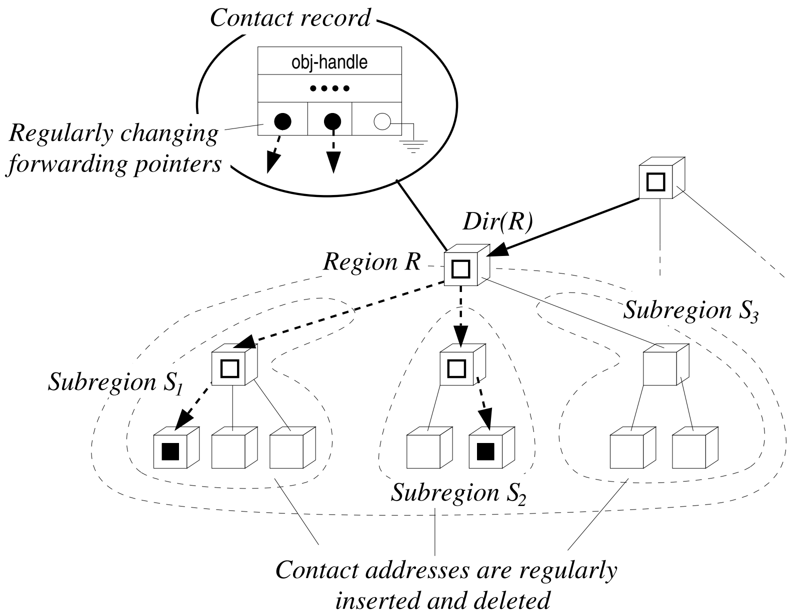

Figure 5: The situation of an object regularly moving between subregions. The solution is to store its changing address in $dir(R)$.

Each time the object moves to a new subregion S_k, the location service creates a path of forwarding pointers from $dir(R)$ to a leaf node in S_k. Likewise, when moving out of S_k the path has to be deleted. If migration occurs regularly, it makes sense to store the address in the object's contact record at $dir(R)$. This not only saves the cost of path maintenance, but more important is that addresses from any of the subregions are now stored at a stable place, namely at the directory node $dir(R)$.

By storing addresses in stable contact records, our model leads to the construction of a search tree *per object*, in which contact records tend to remain in place, even for mobile objects. This permits us to effectively shorten search paths by caching *pointers* to contact records, since they are stable, even if the corresponding objects are not. Specifically, a pointer to the directory node with a contact record containing addresses, is cached at each node of the search path when returning the answer to the leaf node where a lookup request originated.

The combined effect of pointer caches and stable contact records should not be underestimated. An object that moves primarily within a region R can be tracked by just two successive lookup operations: the first one at the leaf node servicing the requesting process, and the second one at the directory node for region R. Moreover, our solution forwards a request in the direction of an address. This is a considerable improvement over existing approaches.

Of course, the migration behavior of an object may change. For example, assume the contact record at $dir(R)$ has contained an address for subregion S_k for quite some time. In that case, the address will be propagated to a directory node in S_k, because apparently, stability occurs in a smaller region than R. Stability is measured by timestamping addresses and forwarding pointers, as well as recording how long an object has not been in a specific region. In all cases, history is taken into account by weighted accumulation of old and new timing information.

Fault tolerance

Any service available in a wide-area distributed system should mask failures of the underlying network to its clients. This is not always possible, because, for example, network partitions may last for hours. Fault tolerant behavior for our location service can be partly expressed in terms of the following informal progress rules:

> **Global progress:** The effect of an update operation initiated at an arbitrary leaf node is eventually visible to a lookup operation initiated at any of the leaf nodes of the search tree.
>
> **Local progress:** The effect of an update operation at a directory node D should be immediately visible at each directory node in the (connected) subtree rooted at D.

The second rule states that an update operation should come into effect at a particular directory node as soon as it is issued at that node, and perhaps before the operation completes. The completion of an operation may depend on the response of the parent node, which may be temporarily unreachable. In other words, update operations are to be handled in an asynchronous fashion. Satisfying the progress rules effectively means that our service is resilient to nodes being unreachable, either caused by network partitioning or because a node has crashed.

To handle operations asynchronously while satisfying the rule for request consistency, incoming requests are first appended to a *queue* and subsequently forwarded to the parent node. Queued operations at a particular node are evaluated in the order they have been appended, and yield a *tentative* result. As soon as the parent agrees with the update, the operation is completed by removing it from the queue and making its result authoritative. If the parent disagrees with the update, for example, when a new address should be stored at a higher-level directory node, the operation is only removed from the queue.

There are several benefits to this approach. First, queuing operations allows us to maintain a consistent, albeit tentative view of the data maintained at a node without having to block any requests until the associated operation is fully completed. This makes the location service resilient to network partitions.

Recovering from node crashes is harder, and is subject to further research. However, a crashed node can easily re-invoke incomplete operations by having its children re-issue their requests, although this is clearly not enough to restore the node's original data set. This approach is very similar to sender-based message logging as discussed in [4], or using queued RPCs as in the Rover toolkit [5]. By replicating authoritative data among the directory nodes in each subtree, the location service can be made resilient against a single node failure in each path originating at the root.

Scalability

Clearly, to construct a worldwide scalable location service it is necessary to adopt hierarchical solutions. For example, nonhierarchical solutions such as the read/write sets proposed in [9] will not do, as it is much harder to exploit locality. However, our search tree described so far obviously does not yet scale. In particular, higher-level directory nodes not only have to handle a relatively large number of requests, they also have enormous storage demands. Our solution is to partition a directory node into one or more **directory subnodes**, such that each subnode is responsible for a subset of the records originally stored at the directory node.

As an example, we can use the first n bits of an object's handle to identify the subnode responsible for that object. Subnodes of a particular directory node need not communicate with each other since they maintain different subsets of objects, and all operations are performed on a per-object basis. Communication between directory nodes in the original search tree takes place only between their respective subnodes. To illustrate, Figure 6 shows a search tree in which the root node has been partitioned into four subnodes based on the first two bits of the object handle ($n = 2$), and each of the leaf nodes into two subnodes ($n = 1$). (We note that we have developed more realistic hashing-based methods than explained here. For example, we also have to take into account that the total number of links between parent and children subnodes remains manageable.)

As communication between directory nodes in the original search tree now takes place between their respective subnodes each subnode should be aware of how the directory node with which it communicates is actually partitioned. This information is contained in a separate tree management service. This service also maintains the mapping of subnodes to physical nodes. Partitioning and mapping information is assumed to be relatively stable, so that it can be easily cached by subnodes. This assumption is necessary to avoid having to query the management service each time a subnode needs to communicate with its parent or children, which would turn the

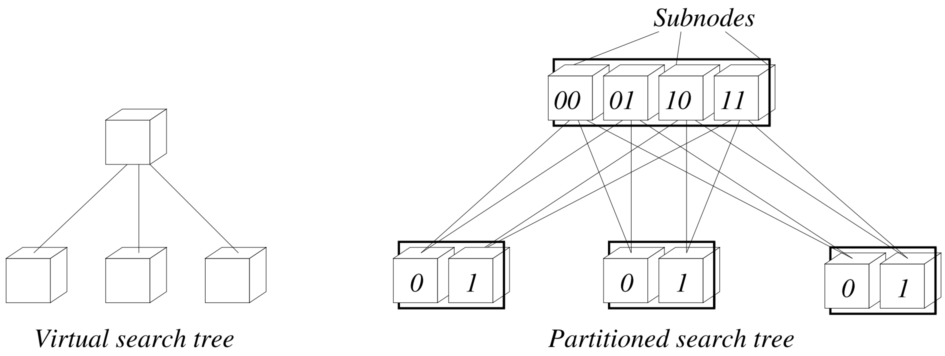

Figure 6: A search tree and a corresponding logical tree after partitioning the directory nodes into subnodes.

management service into a potential communication bottleneck.

4 Related work

Location services are becoming increasingly important as mobile telecommunication and computing facilities become more widespread. So far, mobility is almost invariably connected to *mobile hosts*. A characteristic feature of these hosts is that their mobility is directly coupled to that of their user. This has two important consequences that do not apply to our location service. First, the speed of migration is limited to the maximum speed at which a person can move (typically 1000 km/hour in an airplane), making it possible to adopt a strategy in which data structures gradually adapt as the object moves. This has been successfully applied to several models for location services (see e.g. [1, 7]), but these techniques cannot be used in our case. Second, a host is always at precisely one location. There is no notion of multiple addresses per object as we have introduced in our model. In contrast to current approaches, we can effectively deal with replication.

The situation becomes entirely different when dealing with *mobile software objects*. An important distinction with keeping track of mobile hosts, is that there are many more objects than hosts, immediately leading to a scalability problem. Also, to ensure scalability, it is necessary to take mobility patterns of an individual object into account as well. In Emerald [6], mobile objects are tracked through chains of forwarding pointers, combined with techniques for shortening long chains, and a broadcast facility when all else fails. Such an approach does not scale to worldwide networks. An alternative approach to handle worldwide distributed systems is the Location Independent Invocation (LII) described in [2]. However, LII uses a global naming service as a fallback mechanism, where it assumes that the update–to–lookup ratio is small. Designing a global location service that is not based on such an assumption is an important goal of our research.

A seemingly promising approach that has been advocated for large-scale systems are SSP chains [12]. SSP chains allow object references to be transparently handed over between processes, at the expense of gradually constructing a chain of forwarding references to the object. A serious drawback of this approach is that exploiting locality is completely neglected. In particular, a *home* must keep track where the object is during its entire lifetime, possibly years after the object last lived there. Also, it is unclear how fault tolerance can be efficiently dealt with. SSP chains therefore do not seem to scale to worldwide systems.

Contributions of our approach

One of the main advantages of our approach is that our location service can handle objects that have several contact addresses and that show arbitrary migration patterns. We do not adapt *update and search strategies* to migration patterns, but adapt *the search tree on a per-object basis* instead. By registering contact addresses in the smallest region in which (part of) the object is moving we can make effective use of pointer caches. The combined effect is an extremely short search path, in the optimal case of only length two, from a client to the object. In just two hops it is possible to locate even seemingly randomly migrating objects. This is a considerable improvement over existing approaches.

The use of pointer caches instead of data caches has also been proposed for Personal Communications Systems (PCS). The main reason to apply caching in those cases is to avoid excessive network traffic to the *home location* of a host, which forms the root of a two-level search tree such as in GSM. Caching is done at the second level, by pointing to locations where the host is expected to be found. Cache consistency is achieved either by invalidation on demand, or through active updates. However, caches in PCSs do not account for update patterns. A distinctive feature of our approach compared to PCSs, is that we have several levels allowing us to exploit locality more effectively by inspecting succeeding expanded regions at linearly incrementing costs. On the other hand, locality is also exploited in location updates, making our pointer caches highly effective.

5 Conclusions

The Globe location service offers a novel approach to locating objects in a worldwide context, using location-independent object identifiers instead of user-defined names. Our approach allows an object to update its contact addresses independent of how users have named the object. Locating an object proceeds through a worldwide search tree which dynamically adapts itself on a per-object basis. By storing addresses at stable locations and subsequently caching pointers to those locations, it is possible to contact an object in just two steps, irrespective of the object's migration pattern. An important distinction with current approaches, is that the search path is directed toward the object's present location.

Presently, our research is continuing in two directions. First, we are building a prototype implementation for experimentation and validation of our approach. Second, we are enhancing the basic algorithms described in this paper to include fault tolerance. Also, algorithms for selecting the most appropriate nodes for storing an object's contact addresses need to be further improved.

References

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

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

[3] P. Homburg, M. van Steen, and A. Tanenbaum. "An Architecture for A Scalable Wide Area Distributed System." In *Proc. Seventh SIGOPS European Workshop*, pp. 75–82, Connemara, Ireland, Sept. 1996. ACM.

[4] D. Johnson and W. Zwaenepoel. "Sender-Based Message Logging." In *Proc. 17th Annual International Symposium on Fault-Tolerant Computing*, pp. 14–19, Pittsburgh, PA, July 1987. IEEE.

[5] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. "Mobile Computing with the Rover Toolkit." *IEEE Trans. Comput.*, 46(3):337–352, Mar. 1997.

[6] 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.

[7] P. Krishna, N. Vaidya, and D. Pradhan. "Location Management in Distributed Mobile Environments." In *Proc. Parallel and Distributed Information Systems*, pp. 81–88. IEEE, 1994.

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

[9] S. Mullender and P. Vitányi. "Distributed Match-Making." *Algorithmica*, 3:367–391, 1988.

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

[11] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Sebastopol, CA., 1992.

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

[13] M. van Steen, F. Hauck, and A. Tanenbaum. "A Model for Worldwide Tracking of Distributed Objects." In *Proc. TINA '96*, pp. 203–212, Heidelberg, Germany, Sept. 1996. Eurescom.