Mechanisms for effective caching in the Globe location service

Aline Baggio, Gerco Ballintijn & Maarten van Steen

Vrije Universiteit – Globe Group
De Boelelaan 1081a – 1081 HV Amsterdam
The Netherlands
http://www.cs.vu.nl/globe/

Abstract

Globe is a wide-area distributed system that supports mobile objects. To track and locate objects, we use a worldwide distributed location service, implemented as a search tree.

An object registers its current position by storing its address in a nearby leaf node of the tree. This knowledge propagates up to the top of the tree, so every object can be found from the root. Remote objects can cache the location of an object. However, if the object moves, the cache entry is no longer valid.

In this paper, we show how caching can be made to work effectively even in the presence of mobile objects.

1 Introduction

In recent years, the interest in worldwide mobility and ubiquitous computing has increased considerably. The constantly growing number of mobile phones is one visible aspect of this interest; the common use of laptop computers and handheld devices is another. Furthermore, portable hardware increasingly offers Internet connectivity. For example, it is possible to read one's e-mail from a mobile phone, to browse the World-Wide Web with devices such as 3Com's Palm Pilot [1] or Cyrix's Webpad [5].

Mobile entities such as hosts (i.e. laptops) or software objects usually require special measures for handling both resource discovery and reachability. Resource discovery relates to discovery of nearby services when visiting new networks, such as printers, file or name servers [4, 6], or World-Wide Web caches. Among other things, reachability refers to locating mobile entities, and ensuring location accuracy by means of location updates and/or use of forwarding pointer techniques [13].

One approach to tracking and locating objects is the use of a hierarchical search tree [13, 15, 16]. The tree is built as a hierarchy of domains, that is, geographical or administrative areas, as for DNS. Objects register at a leaf node, but this information propagates up to the top of the tree, leaving a trail of pointers behind. In this way, every object can be located starting at the root, if need be.

Unfortunately, this mechanism can be rather inefficient: to retrieve an object's address, any look-up request may have to go to the root and follow the entire path of forwarding pointers. Efficiency can be improved by caching the result of previous requests. However, with highly mobile objects, the object-to-address mapping is not stable enough to be cached. An alternative strategy that has been proposed for Personal Communication Services (PCS) systems, is that a look-up request can bypass intermediate nodes in the tree by using shortcut links [7].

Rather than only short-cutting the path of a look-up request, we would like to reach the appropriate node directly, that is, where the object's address is currently stored. We call this node the address' *storage location*. Instead of caching an address (i.e. the content), a requesting site will cache a pointer to the address' storage location (i.e. the container of the address).

This scenario works well only when the object remains fixed or migrates within its storage location's domain. If the object regularly moves to other domains, we may need to find another, larger domain that covers the area in which the object apparently always moves. Within this domain, the object may migrate as often as it likes. We call the node responsible for this domain the address' *stable storage location*.

The selection of the stable storage location leads to a tradeoff between stability and accuracy. For example, caching a pointer to a leaf node where an address is stored is accurate but may be unstable. If the object is mobile, its address' storage location will often change, and the cache entry will become invalid too quickly to be actually usable. In contrast, caching a pointer to the root node ensures stability but is inaccurate. Moreover, it unnecessarily puts load on the root node.

In this study, our goal is to find the best node in the tree which satisfies both stability and accuracy concerns. Stability requires finding the lowest node that always knows about the requested object. Accuracy then requires storing the address in this node, so that the requesting site can cache a pointer to it and retrieve the current address.

This paper is organized as follows. Section 2 briefly presents the Globe project, which provides the context of this research, and gives further details about the Globe location service and its requirements for supporting mobile objects. Section 3 shows that stable storage locations can help in efficiently accessing mobile-object addresses, and describes how to detect mobility. Section 4 gives details about the use of storage locations and how to store addresses upwards or downwards in the location service. Finally, Section 5 discusses the related work and Section 6 presents some conclusions.

2 Locating objects

Our research is part of the Globe project. Globe [15] is a wide-area distributed system that provides a scalable infrastructure to support a large number of objects, users and machines spread over the Internet. Among other basic services, Globe provides a location-independent naming scheme. Location independence is achieved by separating proper naming from locating and tracking issues. To this end, Globe distinguishes a naming service and a location service.

These two services use three different ways to designate an object: object name, object handle and contact address. An *object name* is a human-friendly character string [3]. An *object handle* is a unique object identifier. It is independent of the object location. Finally, a *contact address* makes it possible to access the object. It is location dependent: it refers to the physical location of the object, and specifies the communication protocol to use to contact the object. The Globe naming service is in charge of mapping names to object handles, while the location service is in charge of mapping object handles to contact addresses.

The Globe location service is structured as a tree. As in PCS systems, the network is divided into small leaf domains which are aggregated into larger ones. The root domain finally covers the entire network. In the Globe location service, each domain is represented by a directory node. A directory node is in charge of storing mappings between object handles and contact addresses for objects that belong to its domain. To do so, it uses a separate *contact record* for each object (Figure 1). A contact record is created when a new object is registered in the location service; it is deleted when the object is removed.

A contact record is itself composed of *contact fields*, one for each subdomain. A contact field holds either nothing, a set of addresses at which an object can be contacted, or a forwarding pointer. The latter points to the child node responsible for the subdomain where an address for the object can be found. Each contact address can be retrieved by following a path of forwarding pointers from the root to the node where the address is stored. Part of such a path is

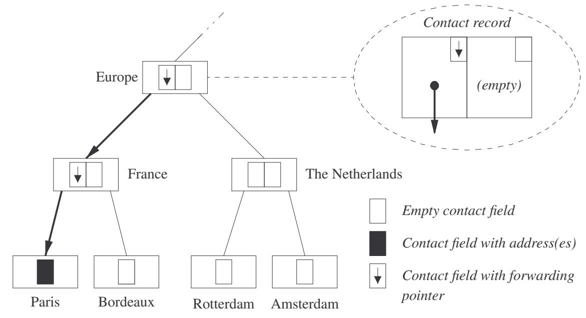

Figure 1: Location service internals

shown in Figure 1, from *Europe* to *Paris*).

To update and use these data, the location service supports requests such as address look-up, address insertion or address deletion. All the invocations (address look-up, insertion, deletion) are initiated at the leaf nodes. For example, if *Rotterdam* in Figure 1 receives a look-up request for the object in Paris, the request will go up to the first node where the object is known (*Europe*). It will then follow the forwarding pointers down to *Paris*, where the address is actually stored.

We saw that in the location service each address has to be reachable from the root. Following any path of forwarding pointers guarantees an address will actually be found. As a consequence, when a contact record becomes empty, the path of forwarding pointers to this contact record is removed.

In our model, a mobile object physically migrates from one computer to another, for example from Amsterdam to Paris. Once this migration is completed, the object is no longer considered present at the old location. To simplify our discussion, we ignore replicated objects in this paper. In the location service, an object migration is seen as an address delete request coupled with an insert request for the same object.

We distinguish two kinds of mobile objects: objects that migrate without disconnecting from the network, and objects that do disconnect. The former are still reachable during migration, such as a mobile phone. The latter become unreachable as soon as they start migrating, such as a laptop computer physically disconnected from any network. Reachability during migration can be obtained, for example, by installing a proxy at the mobile host's "home" location or at its old location, prior to the migration. The proxy will handle incoming messages or requests instead of the real object. For a mobile phone, a proxy could act as a voice mail server.

3 Stable storage locations

Within Globe, the location-service optimization has two aspects. First, by storing a mobile-object's address in a stable storage location, we reduce the number of forwarding pointers to be changed when the object migrates, thus optimizing update operations. Second, by caching pointers to stable locations, we can retrieve an object's address in two hops: one for going to the cache and one to the location that stores the object's address, thus optimizing the look-up operations.

To effectively use location caching, the location service has to detect mobility, select the best stable storage location for a given migration pattern, and actually store the object's address at this location. If the stable location is placed too low in the tree, the object may have moved outside that region when a lookup is done. If it is placed too high, it will be inefficient (having all the European objects in the European node is not practical).

To detect mobility, the location service spies on mobile object migration patterns. It delegates this task to the lowest-level node in which there are always forwarding pointers to the object's address (i.e. the lowest-level node where the contact record is never deleted). This node is the *least common ancestor* (LCA) of the set of leaf nodes in which migrations take place. In Figure 1, for example, if an object frequently migrates between *Paris* and *Amsterdam*, its address will alternately be stored in the *Paris* and *Amsterdam* leaf node, respectively. *Europe* is the lowest-level node (on the path up to the root) from which the contact record is never deleted. It always stores a forwarding pointer to one of its child nodes as there is always an address in one of each. The stable storage location and LCA for this migration pattern is therefore *Europe*.

The LCA is given a monitoring task. It has to record the frequency at which the object migrates. It does so by observing update requests it receives for this object. Monitoring data are kept within the contact record itself. They are composed of creation and modification timestamps, as well as a weighted history of events that is aggregated into a single value. The simple history formulas are as follows, H representing the different history values over time, T the modification timestamps, D the duration between the last two changes and α being the weight parameter.

$$D = T_i - T_{i-1}$$
$$H_i = \alpha * D + (1 - \alpha) * H_{i-1}$$

The history and the timestamp are initialized as follows, $T_{creation}$ being the time at which the contact record was created at the LCA.

$$H_0 = \infty$$
$$T_0 = T_{creation}$$

Whenever the history values decreases below a given *mobility threshold*, the LCA considers the object mobile enough to store its address locally in its own contact record. The mobility threshold is a tolerated frequency of migrations that depends on the level of the LCA in the tree. The higher the LCA is in the tree, the lower the threshold becomes, and the more difficult it is to store an address at that node. What the threshold values should exactly be is still to be determined through experiments.

The modification timestamps are used to calculate the duration between two changes of the contact record. They register changes that occur on different contact fields for the same object. Therefore, if an object migrates between different subdomains of the LCA, the history is regularly updated.

That is not the case any more either if the object does not move any more, or if it moves only in one of the LCA's subdomains. In other words, problems arise if history updates no longer take place. Assume that the LCA stores the current address of an object. A last modification time far in the past indicates that this LCA is no longer appropriate for storing the object's current address. This state triggers the selection a new LCA.

From time to time, the location service therefore has to check whether the last modification timestamp has reached a maximal tolerated value. This leads to the following formula, where T_{now} is the current time and Max the time threshold dependent of the LCA level.

$$T_{now} - T_i > Max$$

Checking the last modification timestamp can be triggered either when an object's address is looked up, or by running a background process. The latter can either be periodic, such as a garbage collection process, or can occur as the need arises, for example if the maximal storage capacity of the LCA is almost reached.

4 Storing addresses upwards and downwards

The location service has to change the storage location according to the object's behavior. This is achieved in two ways. First, by storing an address upwards in the location service tree when an object starts migrating within a broader domain with respect to its current LCA. Second, by storing an address downwards when an object narrows its migration range (and henceforth migrates within one of the current LCA's subdomains), or when the object does not migrate anymore (the object is now fixed).

4.1 Storing upwards

Storing upwards is done using the insert and delete requests that constitute a migration. There are two alternatives: either the location service first deletes the old address and then inserts the new one, or it does the insert first and then the delete.

When the delete occurs first, the contact-record's statistics in the LCA can be lost, as well as the path of forwarding pointers from the root to the LCA. Later on, when inserting the new address, this path will have to be re-established. Preferably, we want to avoid breaking down and later building up again this part of the path of forwarding pointers.

In contrast, if the insert operation takes place first, none of these problems occur. For example, the part of the current path from the root to the LCA is left unaltered. The location service has now simply to ensure that the LCA stores the address instead of any of its siblings. In conclusion, matters are greatly simplified if we can *always* ensure that the insert operation for the new address is carried out before the delete of the old one.

A simple and naive solution consists of starting the insert operation, waiting for it to complete, and only then starting the delete. In a worldwide distributed system, the propagation of the insert request may take a long time. This solution can therefore introduce considerable delays before the delete operation can be carried out.

This solution is appropriate for objects that remain reachable during migration, possibly by means of a proxy. For other types of mobile objects, the location service needs to ensure that the old address is not looked up, as it is no longer valid. Therefore, the location service first invalidates the old address, preventing other objects from using it, and then simply carries out the delete after the insert has taken place. Details are still subject to further study.

It is important to note that when an object migrates outside its LCA's domain, all the statistics gathering and the selection of the stable storage location have to start all over again. The default behavior is then to insert the address in the leaf node of the domain where the object migrated to.

4.2 Storing downwards

Downwards storage is achieved by allowing an LCA's child node to request the addresses of objects that have narrowed their migration range. A child node sends a specific request to its parent node, that is, the LCA, and specifies which address it wants. Upon receipt of such a request, the LCA can decide whether it wants to keep the address itself. If so, the LCA gives the address to its child node and stores a forwarding pointer to it. Otherwise, the request is simply rejected.

The child node has to know when to request an address. To that end, the LCA triggers downwards storage by sending its child node a specific notification. Furthermore, it is the task of the LCA to determine the subdomain, and therefore the child node, where to send both the address and the notification. It is important to note that, later on, the LCA can still reject a downwards storage request it has initiated. This approach is mainly motivated by the need for stateless nodes. The location service nodes, of course, have to maintain their own state including contact records databases, pending requests or tree architecture. However, they are stateless with respect to events happening at their parent or child nodes. For this reason, the LCA will treat the request for an address by its child node, as if it sees it for the first time.

5 Related work

Many experiments have shown that location-based names are not sufficient for locating mobile hosts or objects [2, 6, 8, 9, 10, 11, 12], even if their mobility rate is quite low. Common architectures of distributed location services leads to two approaches: a two-tier scheme and a tree-structured hierarchical scheme.

The first approach (i.e. two-tier schemes) uses home databases located in a predefined network zone. Each mobile entity is assigned both a network zone and a home which becomes permanently responsible for the mobile entity. This home database is in charge of maintaining the current location of the mobile entity up-to-date, and answers location requests. This approach has been used for example with GSM [14] or Mobile IPv6 [9].

The second approach, on which our mechanisms are based, is the tree-structured hierarchical scheme, which have been long used in telephony. The solution proposed in [7] uses such a hierarchical approach for PCS systems. It aims at reducing network traffic when locating mobile entities by using caches as well as shortcut links. It leads to cache pointers to intermediate nodes where the mobile host is known. In this context, caching is used to reduce the length of the path of forwarding pointers to be followed by a search request. In our approach, caching is made more accurate by caching a pointer to the node where the address is currently stored.

6 Conclusion

The purpose of this study is to provide an efficient location service for mobile objects. We consider a distributed search tree, whose nodes store addresses. Optimizing such a location service has two aspects: ensuring a fast and accurate response when looking up objects, and minimizing the changes to be done in the tree when updating addresses (update of forwarding pointers). The use of what we call stable address storage locations helps for both of these aspects.

In the Globe location service, it is relatively simple to identify stable locations. The location service collects statistics about object migrations. Whenever required, it has to ensure that the object's address is stored in the stable location.

The stable storage location is strongly dependent on the object's migration pattern. Each time the migration pattern changes, the location service has to re-evaluate the stable location and ensure that it is still relevant. If not, a new one has to be found, and the address is transferred to that node.

Further research on an effective location service mainly concern experiments. We intend to build a cache for storing pointers to stable storage locations. An optimized look-up operation capable of using the stable storage locations will be necessary as well. We plan to conduct experiments with real traces of object accesses. To that end, we will use access requests from the World-Wide Web and other applications such as "I seek you" (ICQ) tools.

References

[1] 3Com. Palm VII connected organizer, 1999. http://www.3com.com/palm/palm_vii/palm_vii.html.

[2] Awerbuch, B., and Peleg, D. Concurrent on-line tracking of mobile users. In *ACM SIGCOMM Symposium on Communication, Architecture and Protocols* (Oct. 1991).

[3] Ballintijn, G., Verkaik, P., Amade, E., van Steen, M., and Tanenbaum, A. S. A Scalable Implementation for Human-Friendly URIs. Tech. Rep. IR-466, Vrije Universiteit, Department of Mathematics and Computer Science, Oct. 1999.

[4] Bhagwat, P., Perkins, C. E., and Tripathi, S. K. Transparent resources discovery for mobile computers. In *IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, US, Dec. 1994).

[5] Cyrix. Webpad, 1999. http://www.google.com/search?q=cache:www.cyrix.com/html/emerging/webpad/wp_bkgrd.htm.

[6] Forman, G., and Zahorjan, J. The challenges of mobile computing. *IEEE Computer* (Apr. 1994), 39–47.

[7] Jain, R. Reducing Traffic Impacts of PCS using Hierarchical User Location Databases. In *International Conference on Communication* (1996), IEEE.

[8] Jannink, J., Lam, D., Shivakumar, N., Widom, J., and Cox, D. C. Efficient and flexible location management techniques for wireless communication systems. *ACM/Baltzer Science Publishers Wireless Networks 3*, 5 (Oct. 1997), 361–374.

[9] Johnson, D. B., and Perkins, C. Mobility support in IPv6. Internet Draft, Nov. 1998.

[10] Krishna, P., Vaidya, N. H., and Pradhan, D. K. Static and adaptive location management in mobile wireless networks. *Computer Communications (special issue on Mobile Computing) 19*, 4 (Mar. 1996).

[11] Lin, Y. B. Determining the user locations for personal communications networks. *IEEE Transaction on Vehicular Technology 43*, 3 (1994), 466–473.

[12] Mohan, S., and Jain, R. Two user location strategies for personal communications services. *IEEE Personal Communications 1*, 1 (1994), 42–50.

[13] Pitoura, E. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering* (2000).

[14] Scourias, J. An overview of the Global System for Mobile communications. Tech. rep., University of Waterloo, May 1995.

[15] van Steen, M., Homburg, P., and Tanenbaum, A. Globe: A Wide-Area Distributed System. *IEEE Concurrency 7*, 1 (Jan. 1999), 70–78.

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