A Model for Worldwide Tracking of Distributed Objects

Maarten van Steen, Franz J. Hauck, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam

Abstract

We describe a service for locating distributed objects identified by location-independent object identifiers. An object in our model is physically distributed, with multiple active copies on different machines. Processes must bind to an object in order to invoke its methods. Part of the binding protocol is concerned with contacting the object, which offers one or more contact points. An object can change its contact points in the course of time, thus exhibiting migration behavior. We present a solution to finding an object's contact points which is based on a worldwide distributed search tree that adapts dynamically to individual migration patterns.

1 Introduction

The introduction of the World Wide Web and the ease of access to the Internet is radically changing our perception of worldwide distributed systems. Such systems should allow us to easily share and exchange information. This also means that it should be easy to track sources of information, even if these sources move between different locations. (We do not address the problem of finding *relevant* sources of information as is done by, for example, resource discovery services [11].) A key role in tracking sources of information is played by naming systems.

An important problem with current naming systems for wide area networks is that names are location-dependent: a name is tightly coupled to the location of the object it refers to. For example, a URL such as *http://www.ripn.net/nic/rfc/rfc1737.txt* is the name of a Web page containing the text of RFC 1737. The name reflects exactly where the page is stored. If the page is moved or replicated, the name will have to change as well. What is needed is a naming and identification facility that hides all aspects of an object's location. Users should not be concerned where an object is located, whether it can move, whether it is replicated, and if it is replicated, how consistency between replicas is maintained. This mechanism should be available to all applications as a standard facility. Above all, it should scale to the entire world, and be able to handle trillions of objects.

In this paper, we outline a solution for locating objects using location-independent identifiers. Our approach is based on a model in which processes interact and communicate through **distributed shared objects** [5]. Each object offers one or more interfaces, each consisting of a set of methods. Objects are passive; client threads use objects by executing the code for their methods. In order for a process to invoke an object's method, it must first bind to that object. This means that an interface belonging to the object, as well as an implementation of that interface must be placed in the process' address space. To this end, a distributed object has one or more contact points. A **contact point** specifies the network address and protocol with which initial communication with the object can take place. An object's contact points may change in the course of time. For example, an object can be said to expand into, or withdraw from a region when contact points in that region are established or removed, respectively.

We propose a two-level naming hierarchy for finding contact points. The first level deals with hierarchically organized, user-defined name spaces. These name spaces are handled by a distributed **naming service**. However, where traditional name service implementations maintain name-to-address bindings, names in our approach are mapped to object handles. An **object**

handle is a globally unique, and location-independent object identifier. (They have also been coined pure names in [10].) Object handles form the second level in the naming hierarchy. Each object handle is mapped to an object's contact addresses. A **contact address** is a description of a contact point, such as an IP address or the address of the current cell in the case of mobile telephones. It is the task of a **location service** to maintain the mapping between object handles and contact addresses. The design of a possible location server is the subject of this paper.

Our solution comprises a search tree in which an object's contact addresses are stored at relatively stable locations, near to the places where the object can be reached. We show how these stable locations are identified dynamically, and that they may change as the migration behavior of the object changes over time. Storing contact addresses at stable locations permits us to effectively cache location pointers. The combination of dynamically identifying stable storage locations for contact addresses, and caching pointers to those locations, is new. The result is a location service that is highly efficient by exploiting locality in lookup and update operations.

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. We first present the logical organization of our location service in Section 2, and some important optimizations in Section 3. The scalability of our approach is discussed in Section 4. We conclude with a comparison to related work in Section 5.

2 The location service

In our model, we assume a hierarchical decomposition of a (worldwide) distributed system into regions. For example, a lowest-level region may consist of a departmental local area network, whereas a region one level higher may constitute a campus-wide network. This decomposition is relevant to only the location service. It is entirely transparent to client processes.

With each region we associate a **directory node**, capable of storing contact points that lie within that region. This leads to a logical organization as shown in Figure 1. As we explain in Section 4, this organization corresponds to a *virtual* search tree, in the sense that each directory node is distributed across several physical nodes for scalability purposes.

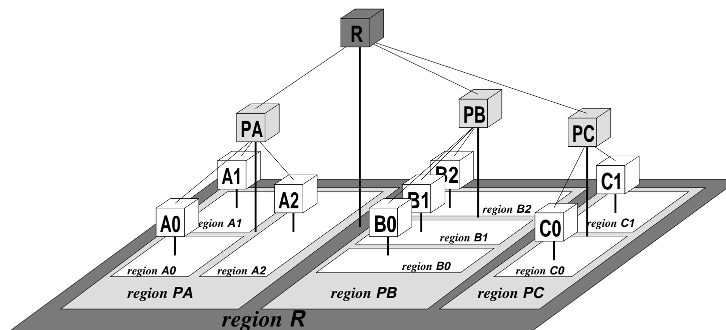

Figure 1: The logical organization of the location service as a virtual search tree.

Each lowest-level region contains one or more distributed objects called **location resolvers**. A location resolver offers an interface to the location service, making the latter appear as just another distributed shared object. Additionally, a location resolver encapsulates locality by allowing clients to transparently communicate with the leaf node for its region. A process is automatically bound to a location resolver.

Inserting and deleting contact addresses

A contact address is initially entered and stored at the leaf node of the tree representing the location of the corresponding contact point. The location service also maintains, per object, a path of forwarding pointers from the root to each leaf node where a contact address is stored. Contact addresses and forwarding pointers are stored in **contact records**. An empty contact record contains only forwarding pointers, whereas a nonempty contact record will contain at least one contact address. 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.

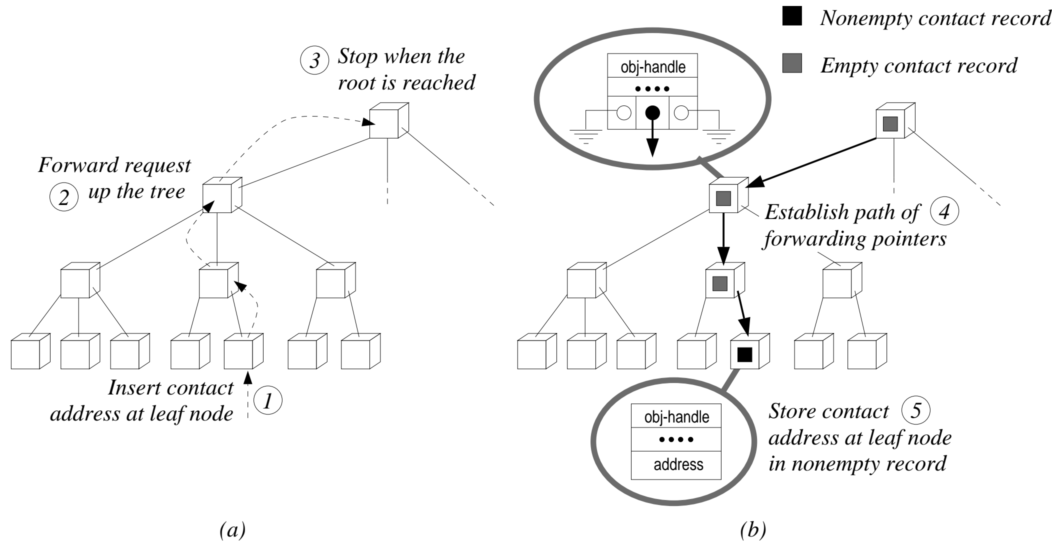

Figure 2: Inserting a contact address for a previously unregistered object, resulting in a path of forwarding pointers from the root to the leaf node.

Figure 2 illustrates the insertion of the first contact address for an object. First, the request for insertion is propagated up the tree to the root, as shown in Figure 2(a). Then, a path of forwarding pointers is established from the root to the leaf node where the insertion takes place. An empty contact record containing a forwarding pointer is created at each intermediate node, as shown in Figure 2(b). The address is finally stored in a record 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. As shown in Figure 3(a), an insertion request propagates upwards to the first higher-level directory node where the object is already known. From there on, a path of forwarding pointers is established to the leaf node where the insertion takes place, as shown in Figure 3(b). 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.

The insert operation returns a **region identifier** identifying the leaf node where the insertion takes place, and which can be used for deletion. 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 identified by the region identifier that was returned when the address was inserted. We also support deletion of contact addresses without knowing the region identifier, by an exhaustive search through the tree. Once the contact address has been found, it is removed from its record. If a contact record contains no contact addresses or forwarding pointers, it is deleted. The parent

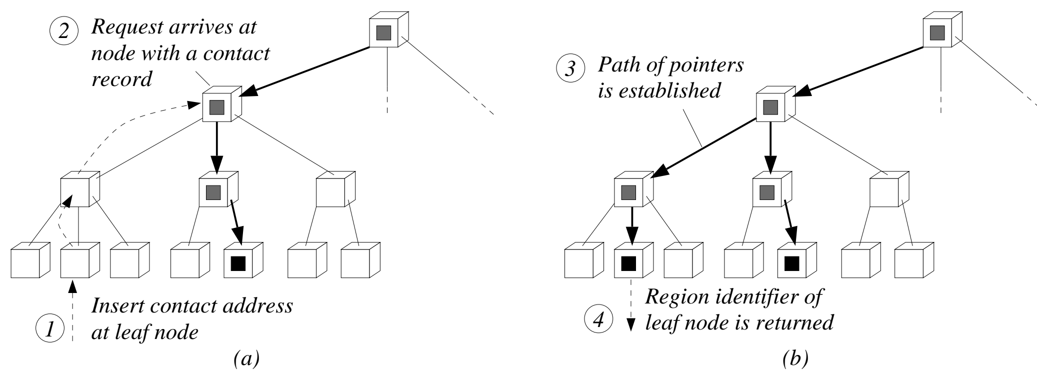

Figure 3: Inserting a contact address when the object is already known. Only the missing pointers are established.

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.

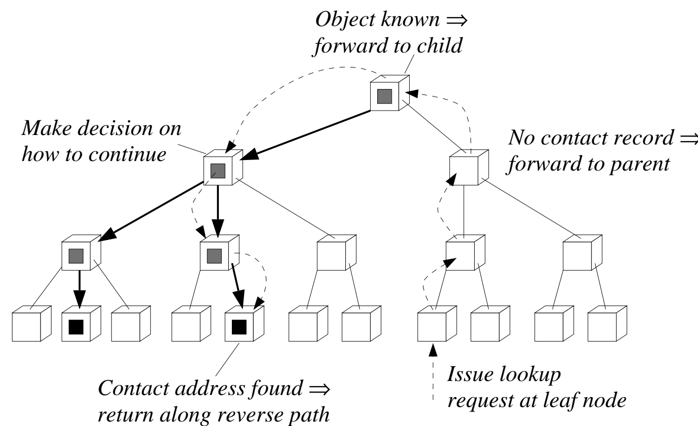

Figure 4: The default approach for looking up a contact address.

It is seen that inserting and deleting contact addresses exploits locality, especially when contact addresses already exist in the region where the operation is being performed.

Looking up contact addresses

Looking up a contact address is done as follows. A process provides its location resolver with an object handle, a quasi-random binary number, which it will often have obtained by means of the naming service. The handle is then given to the leaf node of the resolver's region. As shown in Figure 4, a linear search path is established starting at the client's leaf node, and 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 contact addresses are then returned to the requester.

Note that there may be alternative ways for continuing a search path when a directory node has several forwarding pointers to different contact records. Strategies for selecting amongst alternatives are beyond the scope of this paper. Furthermore, clients can specify a minimum and maximum number of contact addresses that should be returned. The location server then

applies backtracking to find contact addresses. When a nonempty contact record is reached, its contact addresses are aggregated, but the search continues if the minimum number has not yet been reached.

Again, it is seen that locality is exploited: the lookup operation searches local regions first, and gradually expands to larger regions if no contact addresses are found.

3 Optimizations

We now explain how the location service dynamically finds optimal solutions on a per-object basis. The optimizations are based on the observation that caching mechanisms are only truly effective if cache entries change infrequently. For the location service, the only data it has full control over are the placement of contact addresses in contact records (not the placement of the objects themselves). This means that if we can place contact addresses in stable locations, we can make effective use of pointer caches during lookup operations. As we show below, we can even do this for highly mobile objects. Stabilizing the placement of contact addresses and subsequently constructing pointer caches, is a distinctive and novel feature of our approach.

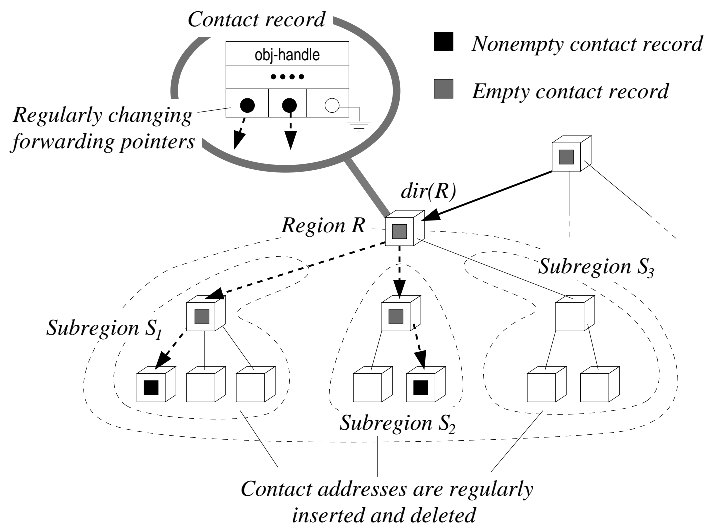

Figure 5: The situation of an object regularly moving between subregions.

Stabilizing the placement of contact addresses

Our first concern is to decide in which contact record a contact address is to be stored. By default, an object's contact 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 contact addresses regularly between the subregions S_1, S_2, and S_3. Also, assume that there is always at least one contact address somewhere in R, so that there will always be a contact record for O at directory node $dir(R)$.

Each time the object expands to a subregion S_k the location service creates a path of forwarding pointers from $dir(R)$ to a leaf node in S_k. Likewise, when withdrawing from S_k the path has to be deleted. If expansion and withdrawal occurs regularly, it makes sense to store the contact address in the object's contact record at $dir(R)$, thus saving the cost of path maintenance. In

addition, addresses of contact points in any of the subregions are now stored in a stable place, namely at the directory node $dir(R)$. This leads to a situation shown in Figure 6.

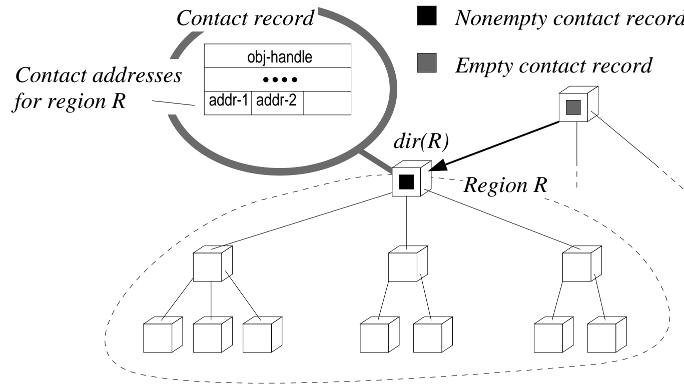

Figure 6: Storing contact addresses in a stable place at a higher level.

Of course, the migration behavior of an object with respect to a region may change. For example, assume the contact record at $dir(R)$ has contained a contact address for subregion S_k for quite some time. In that case, the contact 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 contact addresses and forwarding pointers, as well as recording how long an object has not had a contact point in a region. In all cases, history is taken into account by weighted accumulation of old and new timing information.

Using pointer caches

By storing contact 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 if the associated object moves. This permits us to effectively shorten search paths by caching *pointers* to contact records. Specifically, a pointer to the directory node containing a nonempty contact record is cached at each node of the search path when returning the answer to the leaf node where a lookup request originated as shown in Figure 7.

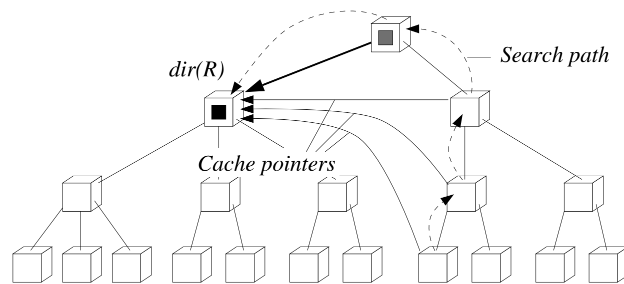

Figure 7: Installing cache pointers after looking up a contact address in the tree from Figure 6.

The combined effect of pointer caches and stable contact records should not be underestimated. An object that primarily moves 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 a contact point. This is a considerable improvement over existing approaches.

Caches are kept consistent in a lazy fashion. A timeout value is associated with each cache entry, which depends on the stability of the referenced contact record. Caches can then, for example, be regularly purged by a sweep algorithm, thus preventing the use of timers per cache entry. In principle, the only other time that a cache entry can be invalidated is when the referenced contact record has been deleted, or when it no longer contains contact addresses.

In the first case, the cache entry is removed and a lookup follows the default strategy described before. In the second case, several strategies are possible. One is to continue the lookup from the directory node containing the referenced contact record. Once contact addresses have been found, the search path is traversed in the opposite direction, thereby updating or establishing cache entries of intermediate nodes. Alternatively, the cache entry can be immediately invalidated after which the default lookup strategy is followed.

4 Scalability and implementation

The search tree described so far obviously does not scale. In particular, higher-level directory nodes not only have to handle a relatively large number of requests, they also have enormous storage demands. For example, the root node needs to maintain a contact record for *every* registered object. This requires already a storage capacity in the terabyte range. In this section we show that our location service scales using a partitioned implementation of the search tree.

Partitioning directory nodes

We partition a directory node into one or more **directory subnodes**, such that each subnode is responsible for a subset of objects. As an example, we can use the first n bits of an object 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 only takes place between their respective subnodes. To illustrate, Figure 8 shows a search tree in which the root node has been partitioned into four subnodes based on the first two bits ($n = 2$), and each of the leaf nodes into two subnodes ($n = 1$).

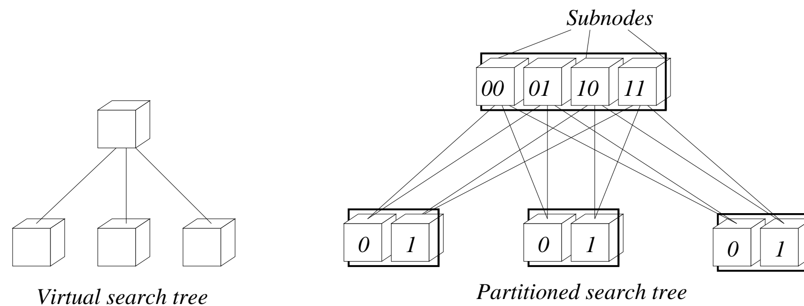

Figure 8: A search tree and a corresponding logical tree after partitioning the directory nodes into subnodes.

In this example, we can guarantee a reasonable load balance if the object handles are uniformly distributed, or otherwise, if the partitioning is based on uniformly distributed numbers, uniquely derived from object handles. This is the case, for example, if the first m bits of an object handle are always generated as a random number and $n \leq m$ for all n.

In principle, each directory node can be partitioned independently according to the number of available hosts and the expected load. In practice, this independence can compromise scalability: each parent subnode may need to maintain a link to every child subnode. Consequently, the number of links between a partitioned parent node and its partitioned children may be prohibitively large. In our example, this problem does not occur. By agreeing to base *all* partitions on the *leftmost* bits of object handles, we have chosen for a common partitioning scheme. However, each node may still individually decide on the number of leftmost bits it will use for partitioning. Without going into further details here, it can be shown that such partitioning schemes can be readily devised. The result is that the number of parent–child links is dramatically reduced and that scalability is not compromised.

Finally, if we assume that the root can be partitioned into 10,000 subnodes (which means that a single root subnode will have to provide service to approximately $10^9/10^4 = 100{,}000$ users), we need also not run into any storage problems. If an object has, on average, five contact points, a contact record at the root will consist of five pointers, the object's handle, and some additional stability information. We expect that this requires no more than 100 bytes. With a total of 10^{12} objects, each of the 10,000 root subnodes will have to store 10 gigabytes of data. This is definitely manageable. Similar results can be derived for the other directory nodes.

Implementing directory subnodes

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 what is called a **meta node** of which there is one per directory node. A meta node also maintains the mapping of subnodes to physical nodes.

Each subnode of a directory node knows where the respective meta nodes of its parent and children can be reached. We assume that the mapping of meta nodes onto physical nodes is reasonably stable. Partitioning and mapping information contained in a meta node is also assumed to be relatively stable, so that it can be easily cached by subnodes. This assumption is necessary to avoid that meta nodes are queried each time a subnode needs to communicate with its parent or children, which would turn the meta node into a potential communication bottleneck.

5 Discussion and related work

We have made a strict separation between a naming service which is used to organize objects in a way that is meaningful to their users, and a location service which is strictly used to contact an object given a unique identifier. Naming services can be used for finding information based on the *meaning* of a name, as is often used for Internet resource discovery services [11]. In our scheme, information retrieval would start with finding relevant names, retrieving the associated object handles, and having the location service return contact address for each object that was found to be potentially interesting.

Location services are particularly important when sources of information, i.e. objects, can migrate between different physical locations. They are becoming increasingly important as mobile telecommunication and computing facilities become more widespread. To relate our work to that of others, we therefore concentrate primarily on aspects of mobility, for which we make a distinction between mobile hosts and mobile objects.

Mobile hosts

So far, much research has concentrated on *mobile hosts*, usually either hand-held telephones or mobile computers. 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 mobility as considered in this paper. First, the speed of migration is limited to the maximum speed at which a person can move: about 1000 miles per hour. Second, a host is always at precisely one location. There is no notion of multiple contact points as we have introduced in our model.

Both features influence the design of a location service. By assuming a speed limit with respect to migration, it becomes possible to adopt a strategy in which data structures gradually adapt as the object moves. This has been used in very different types of location services. For example, mobile hosts in the Internet are supported at the network level where routing tables are dynamically adapted as a host moves [9]. In location services which make use of a distributed search tree, higher-level nodes can effectively cache network addresses rather than pointers to addresses. Whenever a host moves, it leaves a forwarding pointer to its next location. Caches are updated when a considerable distance has been traveled (as described in [1]), or after a lookup. But the speed limitation may even make it possible to avoid the use of caches altogether as illustrated in [14].

Mobile objects

The situation becomes entirely different when dealing with mobile objects that (1) are not statically bound to a single host, and (2) can travel at almost the speed of light. This is the case, for example, with World Wide Web pages and distributed (shared) objects in general.

Mobile objects have mainly been considered in the context of local distributed systems. In Emerald [7], 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]. By combining chains of forwarding references, stable storages, and a global naming service, an efficient mechanism is derived for tracking down objects. Most of the applied techniques are orthogonal to our approach, and can easily be added to improve efficiency. However, the global naming service which is essential to LII assumes that 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]. The principle has been applied to a system called Shadows [3]. SSP chains allow object references to be transparently handed over between processes. In essence, an object reference is just a pointer to a data structure (called a *stub*), that acts as a representative for the object. When passing an object reference from one process P to another process Q, a network connection between the two is established. P's endpoint of this connection is called a *scion*, that of Q is the stub just mentioned. The scion is capable of resolving an incoming reference to the referenced object, possibly by passing it to the stub in its own address space. Consequently, there is no need for any location service because an object reference can always be resolved through the chain of scion-stub pairs that the holder of the reference has established with the object. A serious drawback of this approach is that exploiting locality is completely neglected. This is completely unacceptable for worldwide systems.

Contributions of our approach

One of the main advantages of our approach is that our location service can handle distributed objects that have several contact points and that show arbitrary migration patterns. In contrast to the approach described in [8], 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 points 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 PCNs. 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. Caching is done at the second level, by pointing to locations where the host is expected to be found. Cache consistency is achieved by invalidation on demand [6], but can even be done through active updates [13]. However, caches in PCNs do not account for update patterns. As also observed in [4], exploiting locality in location updates can reduce tracking costs. A distinctive feature of our approach compared to PCNs, is that we have several levels allowing us to exploit locality more effectively by inspecting succeedingly expanded regions at linearly incrementing costs. On the other hand, locality is also exploited in location updates, making our pointer caches highly effective.

References

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

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

[3] S. Caughey and S.K. Shrivastava. "Architectural Support for Mobile Objects in Large-Scale Distributed Systems." In L.-F. Cabrera and M. Theimer, (eds.), *Proc. Fourth Int'l Workshop on Object Orientation in Operating Systems*, pp. 38–47, Lund, Sweden, Aug. 1995. IEEE.

[4] J.S.M. Ho and I.F. Akyildiz. "Local Anchor Scheme for Reducing Location Tracking Costs in PCNs." In *Proc. First Int'l Conf. on Mobile Computing and Networking*, Berkeley, CA., Nov. 1995. ACM.

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

[6] R. Jain, Y.-B. Lin, and C. Lo. "A Caching Strategy to Reduce Network Impacts of PCS." *IEEE J. Selected Areas Commun.*, 12(8):1434–1444, 1994.

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

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

[9] A. Myles and D. Skellern. "Comparing four IP based Mobile Host Protocols." *Computer Networks and ISDN Systems*, 26(3):349–355, 1993.

[10] R.M Needham. "Names." In S. Mullender, (ed.), *Distributed Systems*, pp. 315–327. Addison-Wesley, Wokingham, 2nd edition, 1993.

[11] K. Obraczka, P.B. Danzig, and S.-H. Li. "Internet Resource Discovery Services." *Computer*, 26(9):8–22, 1993.

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

[13] N. Shivakumar and J. Widom. "User Profile Replication for Faster Location Lookup in Mobile Environments." In *Proc. First Int'l Conf. on Mobile Computing and Networking*, Berkeley, CA., Nov. 1995. ACM.

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