A Scalable Location Service for Distributed Objects

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

We describe a service for locating distributed objects using location-independent object identifiers. An object in our model is physically distributed, with multiple copies simultaneously active 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. Finding an object's contact points is the essence of our location service.

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 find information. However, the Web has illustrated that finding information can be hard. In particular, it presents a serious naming and identification problem.

The 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. If the object is moved or replicated, the name has 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 present a model for locating objects using location-independent identifiers. The paper is organized as follows. In Section 2 we explain the system's basic architecture, followed by a description of the operations for finding and updating an object's contact addresses in Section 3. Important optimizations that relate to the scalability of our approach are explained in Section 4. In Section 5 we briefly discuss how a scalable implementation can be obtained. We conclude with a comparison to related work. The restricted length of this paper prohibits detailed descriptions. The interested reader is referred to [5] for further information.

2 System architecture

In our model, processes interact and communicate through **distributed shared objects** [2, 6]. 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. Multiple processes may access the same object simultaneously. Changes to an object's state made by one process are visible to the others. Objects are physically distributed, meaning that active copies of an object's state can, and do, reside on multiple machines at the same time. However, all implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of the object and are hidden behind its interface.

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. Finding an object's contact points is the first step of binding, and is the main subject of this paper.

In order to find contact points, we propose a two-level naming hierarchy. 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, such as a 128-bit binary number. 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 a telephone number or an IP address.) It is the purpose of a **location service** to maintain the mapping between object handles and contact addresses. 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 The location service

In our model for tracking distributed objects, we assume a hierarchical decomposition of a (worldwide) network into regions. 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.

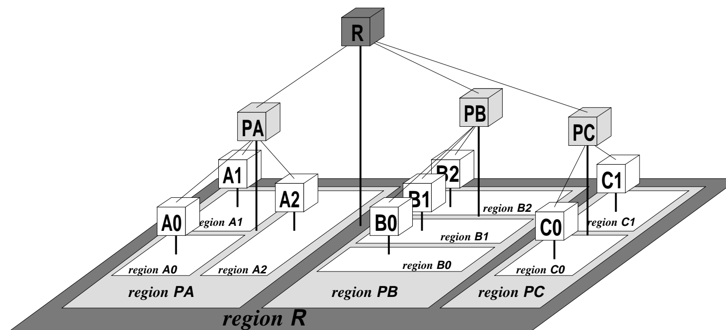

Figure 1: The logical organization of the location service as a virtual search tree.

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.

A request for insertion of a previously unregistered object 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. An empty contact record containing a forwarding pointer is created at each intermediate node. 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 2(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 2(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. Once the contact address has been found,

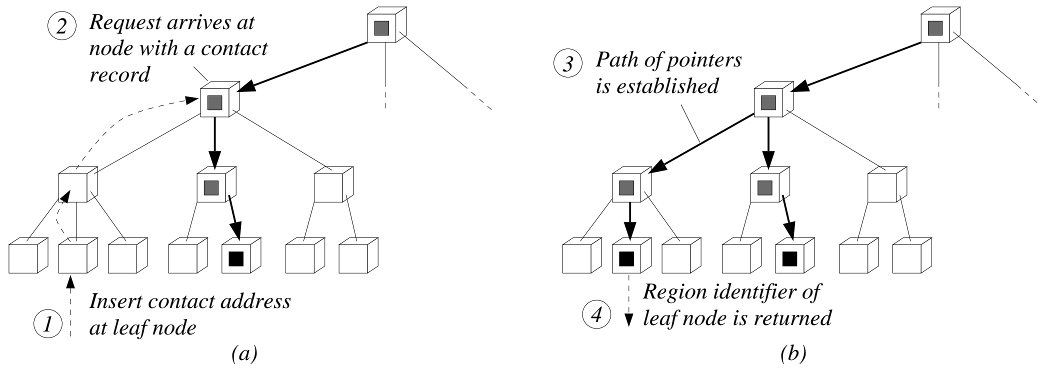

Figure 2: Inserting a contact address when the object is already known. Only the missing pointers are established.

it is removed from its record. If a contact record contains no 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. We also support deletion of contact addresses without knowing the region identifier, by an exhaustive search through the tree.

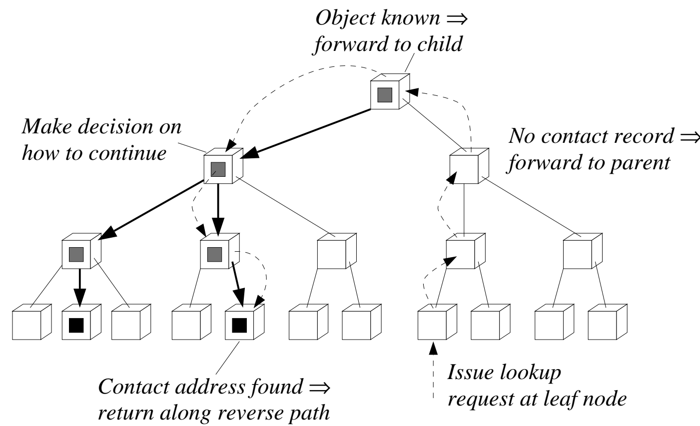

Figure 3: The default approach for looking up a contact address.

Looking up a contact address is done as follows. A process provides an object handle to the leaf node of the region where the process resides. (We require that there is exactly one such leaf node.) As shown in Figure 3, 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.

4 Optimizations

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 4, and assume that an object O is changing its contact addresses regularly between the subregions S_1, S_2, and S_3. For simplicity, 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

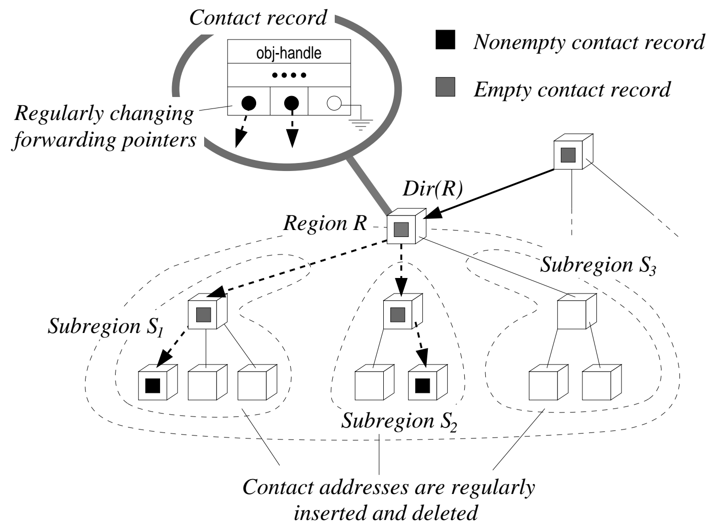

Figure 4: The situation of an object regularly moving between subregions.

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 the situation shown in Figure 5.

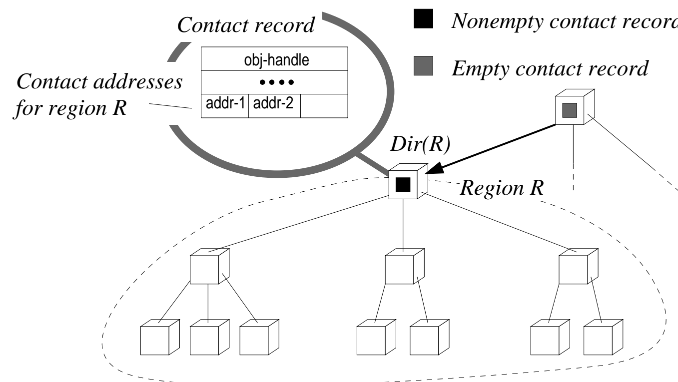

Figure 5: Storing contact addresses in a stable place at a higher level.

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.

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 for mobile objects. 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 6.

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

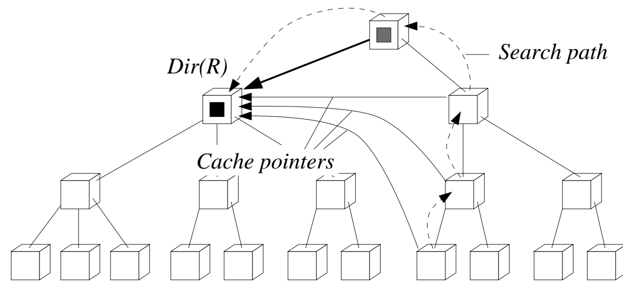

Figure 6: Installing cache pointers after looking up a contact address in the tree from Figure 5.

that a cache entry is invalidated is when the referenced contact record has been deleted, or when it no longer contains contact addresses.

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.

5 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. 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 only takes place between their respective subnodes. To illustrate, Figure 7 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 ($n = 1$).

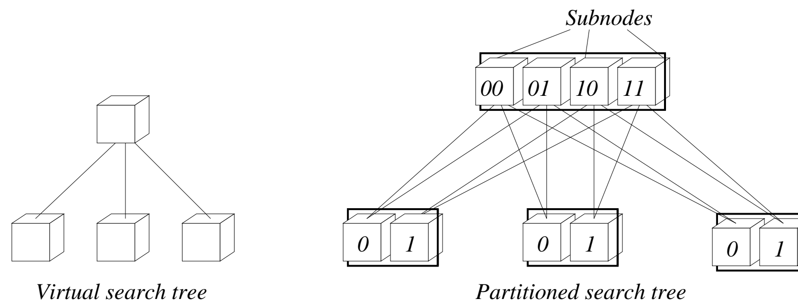

Figure 7: A search tree and a corresponding logical tree after partitioning the directory nodes into subnodes.

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 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 detail 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.

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

6 Discussion and 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. Second, a host is always at precisely one location. There is no notion of multiple contact points as we have introduced in our model.

The situation becomes entirely different when dealing with *mobile objects*. In Emerald [3], 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 [1]. 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 [4]. 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. This is unacceptable for worldwide systems.

References

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

[2] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. “An Object Model for Flexible Distributed Systems.” In *Proc. First Annual ASCI Conference*, pp. 69–78, Heijen, The Netherlands, May 1995.

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

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

[5] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. “A Model for Worldwide Tracking of Distributed Objects.” *Submitted for publication*.

[6] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. “Towards Object-based Wide Area Distributed Systems.” In L.-F. Cabrera and M. Theimer, (eds.), *Proc. Fourth Int’l Workshop on Object Orientation in Operating Systems*, pp. 224–227, Lund, Sweden, Aug. 1995. IEEE.