Locating Objects in a Wide-area System

Gerco Ballintijn

VRIJE UNIVERSITEIT

Locating Objects in a Wide-area System

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 30 oktober 2003 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Gerco Christiaan Ballintijn

geboren te Zaandam

promotoren: prof.dr. A.S. Tanenbaum
prof.dr.ir. M.R. van Steen

Contents

Chapter 1

Introduction

In a large distributed system, locating files and other objects poses a major challenge. This challenge and its solution is the subject of this dissertation. This chapter starts the dissertation with a general description of the location problem and its close relation to naming. It then provides a short description of the Globe project, which is the context of the research presented in this dissertation. It shows how current name services do not meet the naming requirements of Globe, and proposes a new naming scheme. In this new scheme, one of the key challenges is locating resources. The final sections of this chapter identify the research questions for a scheme to locate resources, summarize our main contributions, and provide a road map for the rest of this dissertation.

1.1 Locating Resources in Large Distributed Systems

The current Internet provides its users with many services, including ones that provide information on a wide range of topics. For instance, the World Wide Web (WWW), probably the Internet's most popular application, allows people to find information on topics ranging from world affairs to cooking. The Internet also allows the distribution of free software, and has enabled the emergence of e-commerce, allowing consumers to buy and sell goods over the Internet. Another important service provided by the Internet is the ability for users to communicate with other users. In fact, communication is an integral part of the Internet, with e-mail allowing people to form virtual communities based on shared interests and Internet chat rooms allowing users to exchange up-to-the-minute news.

The last ten years have seen an explosive growth of the Internet. This growth can be seen in both the number of users and the number and range of available information sources and services (from here on collectively referred to as **resources**). Furthermore, the Internet has grown from a network covering only the USA to a network connecting the far corners of the world and everything in between. In short, the Internet has become both a regular part of society and a phenomenon on a worldwide scale.

Also in this period, portable computers, such as laptops, have initiated a new trend in computing: the support for **mobile computing**. Computers are no longer stationary objects with a fixed connection to the Internet. Instead, they accompany users on their travels, allowing them to use their functionality anywhere and at anytime. This mobility is visible both with users connecting their portable computers to the Internet over different dial-up connections and the increasing availability and use of wireless network technology.

Unfortunately, the large scale use of many resources has also introduced **scalability problems**. With many users wanting to use the resources on the Internet, servers that provide these resources are in danger of becoming overloaded. They simply cannot handle the huge number of requests for these resources. Also, while it is possible on the Internet to retrieve information from the other side of the world, this is frequently undesirable given the inherent delay in communication over such distances.

To solve both these scalability problems, resource providers have started to **replicate** resources at multiple locations in the Internet. Replication allows different users to use different replicas of the same resource, thereby spreading the load over multiple servers. Furthermore, by spreading the replicas over the Internet, replication also enables users to use a nearby replica of the resource, reducing communication latency and limiting the load on the network.

To communicate with a resource on the Internet, a user needs to know the network **location** of the resource (i.e., its IP address). Unfortunately, since resources no longer exist at a fixed location with the introduction of mobility, we cannot be sure a resource will be available at the same location in the future. Likewise, since resources no longer consist of a single copy with the introduction of replication, we also cannot be sure that the current nearest replica remains the nearest replica in the future. What we therefore need is a way to track the locations of the replicas of a resource, or preferably, the location of the *nearest* replica.

Location tracking is traditionally done by general purpose **name services**, such as DNS [Mockapetris, 1987; Albitz and Liu, 1992]. These name services can store a wide variety of information on a named resource, including its location. By allowing users to refer to a resource by a name instead of its location, these name services shield users from the problems of where the resource is located, whether it consists of more than one copy, and whether it can move. In other words, name services provide **location transparency**, **replication transparency**, and **migration transparency**, respectively.

Unfortunately, current name services are not equipped to deal efficiently with huge numbers of mobile and replicated resources distributed worldwide. Therefore, to enable more applications to use replicated resources and mobile hardware, we desire a new location tracking service on a worldwide, Internet scale. In this dissertation, we examine this problem of tracking and locating huge numbers of mobile and replicated resources in a worldwide network. We are specifically interested in finding the location of nearby resources where possible. The solution proposed in this dissertation is the Globe location service.

1.2 The Globe Project

The Globe location service is implemented as part of the Globe project [Homburg, 2001], but its design is applicable outside of Globe. The name Globe stands for GLobal Object Based Environment. The goal of the Globe project is to develop a software framework to ease the building of wide-area distributed applications, such as the World Wide Web. The design goals of the Globe project are ambitious. One goal is to support up to 10^{12} resources. This number corresponds to 10^9 clients with each having 1,000 shared resources on average. Another goal we desire is that the Globe software supports resources distributed worldwide. As a consequence, the Globe software must avoid communication over long distances where possible.

To achieve these goals, the Globe model provides comprehensive support for replication. A resource in Globe consists of (potentially) multiple replicas. These replicas are located on different hosts distributed across the network. An application uses the resource by communicating with one of its replicas. To simplify the Globe software model, however, the replicas work together to provide the illusion of simply using a single object. In Globe, we therefore consider the set of replicas to constitute a single, *virtual* **object**. The set of replicas that implement an object is not fixed during its lifetime. For instance, resource providers can add new replicas to an object to provide extra load capacity and remove existing ones when their functionality is no longer required.

Globe also uses a naming system to provide location, replication, and migration transparency. Every resource (and thus object) has an **object name**, and users refer to resources using these object names. To use a resource, a user application gives the Globe naming system the name of the resource, and the naming system selects the replica the application should communicate with. To enable efficient communication, the naming system takes special care to select a replica that is near the host running the application.

An object is called mobile in Globe when it frequently changes its associated (network) location. A location change happens either because the object itself moves from one host to another, for instance, if it implements a mobile agent [Harrison et al., 1995], or because the object is located on a host that migrates from one network to another (e.g., a laptop). The Globe naming system does not distinguish between both forms of mobility, however, since they both simply result in a new locations being associated with the name of the mobile object.

1.3 Naming in Globe

In Globe, object names refer to resources implemented as Globe objects. Since object names are used by people, they must be human friendly. Human-friendly names are usually easy to construct and remember. They frequently carry semantic information to improve their usefulness. Object names are often location independent since the location of a named object is generally irrelevant to a user.

Since different users might have different associations with a resource, they may want to use different names for the object implementing that resource. To support this goal, a Globe object can have multiple names. This support is comparable to WWW users having a personal list with bookmarks for WWW resources they found earlier. Each entry in a bookmarks list has a name chosen by the user for easy remembrance.

To use a resource, a user application, such as a Web browser, communicates with one of the replicas implementing the resource. To perform this communication, the application needs to know the network locations of at least one of the replicas. In Globe, the network location of a replica is described by a **contact address**. An example of a simple contact address is an Internet address, which is the combination of IP address and port number. Given the fact that a contact address describes a location in the network, contact addresses are inherently location dependent. In fact, throughout this dissertation, the terms replica location and contact address can be considered synonymous.

Since object names are used to refer to objects but contact addresses are used to actually communicate with the object, we need to translate the object name into a contact address of one of the replicas of the named object. Providing this translation is the responsibility of the naming system in Globe. Since an object can have more than one name and more than one replica, the naming system must support an **N-to-M** mapping of object names to contact addresses. Moreover, this mapping must be frequently changeable since a mobile object changes its contact address every time its moves its location.

Since the naming system is an integral part of Globe, it has the same scalability requirements as Globe as a whole. The naming system should thus be able to store the contact addresses of up to 10^{12} objects. Furthermore, since an object can have replicas distributed throughout the Internet, contact addresses are distributed worldwide. To remain scalable, however, the Globe naming system must avoid long distance communication where possible. We refer to this goal as “exploiting locality.”

In the Globe system, we can define **locality** using a number of metrics. Some of these metrics are network-oriented, such as the number of routing hops or the communication latency, but we can also use a metric based on, for instance, geographical distance. Since all these metrics can be useful in the naming system at different times, we cannot simply choose one over the other, instead we chose to let the naming system to define its own locality metric. The only requirement for this metric is that it should be both derived from and provide a reasonable estimate of the example metrics. If other distance metrics become important in the future, these should be incorporated as well.

The naming system can exploit locality in several ways. For instance, when requesting a contact address of an object, a user is interested in the address of a nearby replica. Furthermore, we would like the naming system to use only nearby name servers to find nearby contact addresses, that is, the process of retrieving the contact address of a nearby replica should not involve servers located at the other side of the world. Nearby name servers should also be used when updating the set of contact addresses of an object. For example, if a replica moves between two nearby locations, the naming system must use only servers near these two locations for the update operation. As a consequence, the naming system needs to have its name servers distributed throughout the network.

Table 1.1: Main requirements of the Globe naming system.

Provide human-friendly naming
Provide an N-to-M mapping
Support for replication
Support for mobility
Support for 10^{12} objects
Support for objects distributed worldwide
Support for exploiting locality

1.4 Existing Naming Systems

Table 1.1 summarizes the main requirements of the Globe naming system. Unfortunately, existing naming systems do not (completely) fulfill these requirements. This section summarizes four types of existing systems that allow users to track and query the current location of an object and discusses their problems. Note that Chapter 9 provides a more thorough description of related work.

Name and directory services

An obvious candidate for a worldwide naming system for Globe is an existing name or directory service, such as the Domain Name System (DNS) [Mockapetris, 1987; Albitz and Liu, 1992]. DNS is the Internet's name system, and scales successfully to millions of resources. DNS is a good candidate because it has a proven ability to store information, such as IP addresses (i.e., network locations), for a large number of hosts. For instance, as of January 2002, DNS stores 1.5×10^8 names [Internet Software Consortium, 2002].

Host names and e-mail destinations in DNS are collectively known as **domain names**. Domain names are organized into a tree-shaped name space, similar to present-day file systems. A domain name consists of a sequence of labels separated by the dot (“`.`”) character. The sequence represents a path through the name space starting at a common root, called the root domain.

DNS resolves a domain name by visiting a sequence of name servers with each name server able to resolve more labels than the previous one. For instance, to resolve the domain name `www.intel.com`, DNS starts at the server for the root domain, and, in turn, visits the name servers responsible for resolving names ending in `.com` and for names ending in `.intel.com`. The latter name server resolves the complete host name `www.intel.com` into an IP address.

DNS attains its scalability through the use of partitioning, replication, and caching. DNS partitions its name space into a large number of disjoint **zones** (i.e., sets of related domains) and distributes these zones over separate name servers. DNS increases its availability and fault tolerance by replicating the information in a zone over multiple servers. To improve the efficiency of name resolution, name servers also cache intermediate and end results of name resolution operations. To ensure replication and caching is effective, however, DNS requires the domain name to address mapping to change only infrequently.

While DNS can store location information on a large number of resources, it is unclear whether DNS can support the required 10^{12} named resources. This number of names requires DNS to store 10,000 times as much (location) information as it currently does. One source of problems with this many domain names is the average frequency with which a domain name is looked up. If almost all domain names are looked up only infrequently, caching can no longer be effective since the intermediate and end results that are cached are no longer reused by subsequent name lookups. It is unclear whether the average frequency of lookups on a domain name in DNS with 10^{12} names is large enough to provide a similar amount of reuse of cached results as found in DNS with only 10^{8} domain names.

The main source of problems when using a name services, like DNS, in Globe, however, is the requirement to support *frequent* mobility. Frequent mobility requires DNS to frequently update its name to address mapping. These frequent updates render both the caching and replication in DNS inefficient since these updates invalidate cache entries and require communication to propagate to all affected name servers. Directory services, such as X.500 [Radicati, 1994], are equally unsuitable for use in Globe since they too rely on replication and caching to provide scalable naming.

A partial solution to the problem of supporting frequent mobility is disabling the caching and replication of location information for only those resources that are frequently mobile. This approach ensures that the location information of a mobile resource is stored in only a single name server. The approach turns DNS, however, into a home-location based system, with all the associated problems of exploiting locality, as is described next.

Home-location based systems

Home-location based systems are a straightforward solution to the location problem, and are used in many distributed systems, for instance, Locus [Butterfield and Popek, 1984] and MOS [Barak and Litman, 1985]. In a home-location based system, each mobile object is associated with a single server (i.e., its home location) that is responsible for maintaining the location of the object. For efficiency, the location of the server is usually part of the object name. When a process wants to use the mobile object, its current location is looked up at the home location; and when the mobile object moves, its location is updated at the home location. Home-location based systems easily support large numbers of mobile objects since object locations are not centrally stored, and new home-locations can easily be added.

Unfortunately, home-location based systems cannot easily exploit locality, which makes them unsuitable for use in a worldwide naming system. Since the current location of an object is available only at the home location, processes located far from the home location may have to communicate over long distances to find the object's current location. The same applies when an object moves and its location needs to be updated at the home location. When a mobile object is located far from its home location, all communication needed to keep the home location up-to-date is over a long distance. Furthermore, home-location based systems usually have no support for replication. A more general problem is the fact that home-location based systems have no support for human-friendly naming.

Pointer-chain based systems

Pointer-chain based systems have been successfully used in local-area systems that support object mobility, such as Emerald [Jul et al., 1988], and have been proposed for use in wide-area systems, such as Location Independent Invocation (LII) [Black and Artsy, 1990] and SSP chains [Shapiro et al., 1992].

In a pointer-chain based system, clients refer to an object by referring to its last known location. To ensure clients can keep in contact, an object leaves behind a forwarding address when it moves to a new location. When an objects moves past several hosts, it leaves behind a trail of forwarding addresses (i.e., a chain of pointers) ending in its current location. To avoid the inefficiencies associated with long chains of pointers, a chain-reduction mechanism is employed, usually as part of a remote invocation mechanism. Since the (potentially long) chains are susceptible to crashes of nodes in the chain, a fall-back mechanism, such as broadcasting or a name service, is used to increase fault tolerance.

Pointer-chain systems are unsuitable for the Globe naming system for a variety of reasons. The most significant problem is that pointer-chain based systems lack the locality needed in a wide-area system. Since the nodes on the chain of forwarding pointers can be located everywhere, communicating with an object can take a request all over the world before ending up at the object's current location. Furthermore, ensuring both the connectivity and regular reduction of the chain of pointers is a complex task. The techniques used to ensure availability and reduce chains limit the scalability of pointer-chain systems even further. Finally, like home-location based systems, pointer-chain based systems lack support for replication and human-friendly naming.

Search-tree based systems

Location tracking systems also play an important role in cellular mobile phone systems. While the currently existing mobile phone systems use home-location based tracking systems, researchers, such as [Wang, 1993], have proposed to use location tracking systems based on a distributed search tree for next generation mobile phone systems.

In this approach, the area covered by the location system is divided into a hierarchy of service areas. The lowest level of the hierarchy is formed by leaf service areas that record the location of locally present mobile phones. A search tree is formed by combining the small leaf service areas into larger service areas until there is a root level covering the whole service area. Every service area forms a node in the search tree that stores information about which mobile phones are present in that area.

There are two main problems with this approach. The first problem is that the scheme focuses on mobile phones only, and it thus does not support replication. The second

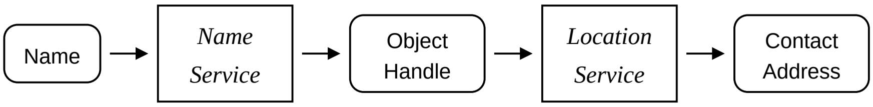


Figure 1.1: Name resolution in Globe consists of two steps: first the naming step, then the location step.

problem is that the scheme does not support human-friendly naming, that is, it does not allow users to easily refer to and identify mobile phones. These problems can be solved, however, as the rest of this dissertation shows. In fact, the location service proposed in this dissertation also uses the distributed search tree approach. Our research can thus be considered to run parallel to research on locating mobile phones.

1.5 A Two-level Naming Scheme

To simplify the naming and location problem in Globe, we divide the problem into two separate problems: object naming and object location. Using this division, we can use a separate **name service** to identify objects and a separate **location service** to identify the current location(s) where an object resides. By using separate services for both problems, we can optimize the name service for naming objects in a human-friendly fashion and optimize the location service for locating nearby replicas of objects.

Name resolution in Globe thus becomes a two-step process, as shown in Figure 1.1. In the first step, the name service resolves an object name to an object handle; in the second step, the location service resolves the object handle to a contact address. An **object handle** is an additional type of name that is used solely to identify objects, thereby allowing us to combine the functionality of naming and location services.

In the Globe naming scheme, object handles play the important role of **proper identifiers** [Wieringa and de Jonge, 1995]. Proper identifiers have the following properties:

- Each object handle refers to exactly one object.
- Each object has exactly one object handle.
- An object handle is never reused.
- An object keeps its initial object handle.

As a consequence of these properties, once a user has obtained the object handle of an object, it can use the object handle as a reference for as long as it desires, without having to worry that the reference might become outdated or that it might refer to another object. The object itself might cease to exist, however.

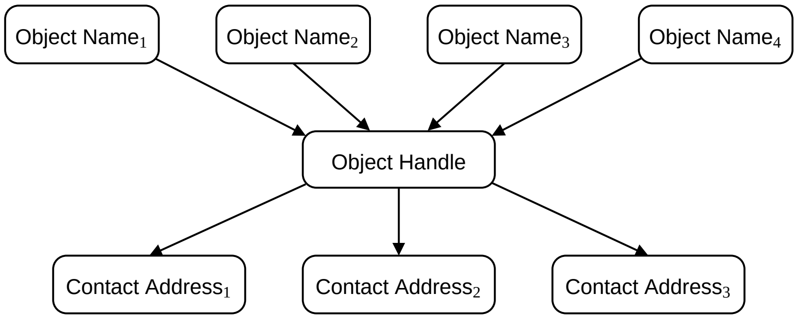


Figure 1.2: The two-level naming scheme used in Globe to map multiple object names to multiple contact addresses. In this scheme, the dynamic N-to-M mapping is split into a stable N-to-1 mapping and a dynamic 1-to-M mapping.

The introduction of the object handle allows us to effectively separate the dynamic N-to-M mapping of object names to contact addresses into a relatively stable N-to-1 mapping (stored by the name service) and a dynamic 1-to-M mapping (stored by the location service), as shown in Figure 1.2. We expect the object name to object handle mapping to be relatively stable since mobility affects only the object handle to contact address mapping.

Since the object name to object handle mapping is expected to be stable, we can use existing name services to store this mapping. They can easily replicate and cache the object name to object handle mapping. The problem of maintaining the dynamic 1-to-M mapping of the location service, however, is not as easily solved. This dissertation thus ignores the problem of naming objects in a human-friendly way, and focuses solely on the problems of the location service.

It is the task of the location service to track the current locations of all (available) replicas of all objects in Globe. Users should be able to query the location service and retrieve the contact address of a nearby replica of an object using the object's object handle. The main functionality of the location service is thus to look up one or more contact addresses of an object handle. To ensure the currency of the set of contact addresses associated with an object, the location service must update the set every time a replica is added, deleted, or moved.

The scalability requirements of the Globe naming system, described in Table 1.1, apply also directly to Globe location service. The location service must support at least 10^{12} object handles, and contact addresses can refer to network locations all over the world. Furthermore, as indicated in Section 1.3, to exploit locality, the location service must consist of servers placed near its clients.

1.6 Research Questions

The general research question examined in this dissertation is:

How can we build a worldwide location service?

What are the problems we encounter while designing such a service, and how can we solve them? To answer this general question, we divide it into several more-specific research questions:

1. *What architecture combines scalability with flexibility?*
   The architecture of the location service has to be scalable to allow the service to be scalable as a whole. Furthermore, since finding solutions to specific problems, such as fault tolerance, is difficult, we might need to try several solutions. Therefore, the architecture has to be flexible enough to support different solutions because once the architecture is established, it will be difficult to change.

2. *How do we avoid centralized components in our architecture?*
   Centralized components are potential scalability and reliability bottlenecks and should thus be avoided.

3. *How do we ensure availability in a huge system such as our location service?*
   The location service plays a pivotal role in communication with objects but is vulnerable to faults given its large size. The service must therefore mask the occurrence of faults in the components of the service.

4. *What kind of security is needed for the information stored in the location service and how do we provide it?*
   Given that the location service plays a pivotal role in communication with objects, it is also a prime target for attack.

5. *How do we ensure that the location service can deal with changes in its environment?*
   Once in use, the location service might exist for a long time (e.g., DNS is well within its second decade). During this time, significant changes will surely occur in the operating environment of the location service. To deal with these changes, the location service should be adaptable, preferable without interrupting its service.

1.7 Contributions and Outline of this Dissertation

The general contribution of this dissertation is a comprehensive design of a worldwide location service. The design is comprehensive in the sense that it covers all critical aspects of a location service. This general contribution can be subdivided into several more-specific contributions based on the specific research questions posed in the previous section. We describe our specific contributions per chapter.

Chapter 2 — Architecture Chapter 2 introduces the architecture of the Globe location service, which is based on a distributed search tree. Its working is explained by providing simplified versions of the algorithms used to lookup and update location information in the search tree. Enhanced versions of these algorithms are provided in later chapters. This chapter provides the conceptual framework needed for the rest of the dissertation.

Chapter 3 — Load Distribution Chapter 3 extends the conceptual model of the distributed search tree, introduced in Chapter 2, by providing a method to deal with a potential scalability problem associated with the root node of the search tree. The effectiveness of the method is evaluated using simulation experiments, which show that we can solve the scalability problem of the root node.

Chapter 4 — An Efficient Lookup Operation Chapter 4 describes how the efficiency of lookup operations can be improved, especially in the face of mobility. It shows why normally caching does not work in the location service, and provides an alternative form of caching. The effectiveness of this new form of caching is evaluated using simulation experiments, which show that our alternative form of caching improves the performance of lookup operations. The research described in this chapter was done in cooperation with Aline Baggio.

Chapter 5 — Availability and Fault Tolerance Chapter 5 describes how the locations service deals with various types of failures, thereby ensuring high availability. This chapter focuses specifically on a crash recovery method that masks and corrects inconsistencies in the distributed state of the location service that were caused by failures.

Chapter 6 — Security Chapter 6 describes our security goals in the location service, and the techniques employed to achieve them. It specifically focuses on the problem of securing communication between nodes in the search tree and preventing unauthorized clients from performing update operations. The research described in this chapter was done in cooperation with Bogdan Popescu.

Chapter 7 — Tree Management Chapter 7 examines some management issues of the location service, and provides an outlook on methods to adapt the distributed structure of the search tree in a changing environment. The research described in this chapter was done in cooperation with Aline Baggio and Spyros Voulgaris.

Chapter 8 — Prototype Implementation Chapter 8 describes a prototype implementation of the Globe location service. It focuses on some preliminary measurements of its performance, and examines these results in the light of the scalability requirements of the location service. The examination of the prototype performance shows that a scalable location service can be built.

Chapter 9 — Related Work Chapter 9 describes eight systems related to the Globe location service. It identifies several categories of related work based on similarity in functionality and the techniques used, and compares typical examples of these categories using the issues raised in Chapters 2–7.

Chapter 10 — Summary and Conclusions Chapter 10 concludes this dissertation. It provides a summary of our work on the Globe location service, and presents some global observations. The chapter finishes with our thoughts on future work.

Chapter 2

Architecture

This chapter describes the basic design of the Globe location service. It starts with a description of the interface provided by the location service, and introduces some design principles that are used to guide the location service design. This chapter then presents the main data structures of the Globe location service, and shows how simplified lookup and update operations use these data structures. Later chapters enhance the data structures and operations to implement the complete functionality of the Globe location service.

2.1 Interface

The Globe location service provides its clients with an interface consisting of three operations: **lookup** one or more contact addresses, **insert** a contact address, and **delete** a contact address. The signature of these three operations is shown in Listing 2.1. To ease our description of the location service interface, the listing does not include a mechanism for communicating error conditions, such as return values or exceptions, which will obviously be present in a real-life implementation.

The *lookupAddresses* operation retrieves the contact addresses of one or more replicas near the client. It is in general useful to retrieve more than one contact address during a lookup operation because a nearby replica might not support the same communication protocol as the client or because the replica might be temporarily unavailable due to a network failure. Allowing the *lookupAddresses* operation to look up more than one contact address at once avoids the overhead of multiple invocations of the *lookupAddresses* operation. Since we expect the desired number of contact addresses to vary among different types of client applications, we let the client specify this number instead of the location service.

The client specifies the number of desired contact address using the *min* and *max* parameters. The *min* parameter specifies the minimal number of contact addresses guaranteed to be returned if available. If fewer contact addresses are available, only those will

```
procedure lookupAddresses(oh : ObjectHandle; min : Integer; max : Integer)
            returns set of ContactAddress;
procedure insertAddress(oh : ObjectHandle; addr : ContactAddress);
procedure deleteAddress(oh : ObjectHandle; addr : ContactAddress);
```

Listing 2.1: Client interface of the Globe location service.

be returned. The *max* parameter specifies the maximal number of contact addresses the client can, or wants to, handle. When the location service has found a number of contact addresses between *min* and *max*, it can determine itself whether to continue searching or not, taking into account, for instance, the amount of extra work involved.

The *insertAddress* operation adds a new contact address to the set of contact addresses associated with an object handle. This operation is invoked when a new replica is added to the specified object. When the *insertAddress* operation is finished, the location service guarantees that subsequent lookup operations can find the new contact address, assuming no failures occur. The *deleteAddress* operation removes a contact address from the set of contact addresses associated with an object handle. This operation is invoked when an existing replica is removed from the object. When the *deleteAddress* operation is finished, the location service guarantees that subsequent lookup operations can no longer find the contact address. Note that, throughout this dissertation, the insert and delete operations are together referred to as **update** operations.

Mobility is supported in the location service by the combined use of the *insertAddress* and *deleteAddress* operations. When a mobile object moves from one location to the next, the *insertAddress* operation is used to inform the location service that the object is present at its new location and the *deleteAddress* operation is used to inform that the object is no longer present at its old location. The order in which the *insertAddress* and *deleteAddress* operations are invoked depends upon whether the mobile object stays available at its old location during its move. If the object stays available, the new contact address is inserted before the old one is deleted. If the object is unavailable during its move, however, the old contact address is deleted before the new one is inserted.

The semantics chosen for the *insertAddress* and *deleteAddress* operations is to make their changes immediately visible. We chose this strict semantic to decrease the load placed on the location service by lookup operations. If we would have chosen a more relaxed semantic, for instance, by guaranteeing that a contact address would only eventually disappear after a *deleteAddress* operation, the users of the location service would be faced with a large number of outdated contact addresses in the case of highly mobile or highly replicated objects. These outdated entries, in turn, would require users to invoke the *lookupAddresses* operation with large *min* and *max* values in the hope of finding at least one valid contact address. By providing our strict semantics, and thereby minimizing the risk of finding outdated contact addresses, users have to look up only one or a few contact addresses.

Our object model assumes that the set of replicas implementing one object is completely independent of the set of replicas implementing another object. The associations between object handles and their contact address sets are therefore also independent. The location service can thus handle operations on different object handles independently, that is, an insert operation for one object does not interfere with, for example, the delete operation for another object.

2.2 Design Principles

From the general goal to design a worldwide location service that supports up to 10^{12} objects, we have extracted some design principles to achieve that goal. These design principles are variants of a single basic rule: Minimize everything that uses system resources, such as storage, processing, and communication. The central notion is that since the location service already has to use all the available resources to support a huge number of objects across a huge geographical area, we should avoid adding other features that might also require these resources.

Locality

As described in the previous chapter, a primary design principle is to exploit locality in the location service where possible. This means that we want to store and process location information, such as contact addresses, near the users of that information. For instance, if a client looks up a contact address of an object and the object has a replica in the vicinity of the client, the location service should return the contact address of that replica; furthermore, the process of finding the address should not require communication with hosts located far away.

Our design is therefore guided by two forms of locality: *externally visible* locality, which is finding a nearby replica; and *internally visible* locality which is using only nearby location servers to find a replica. Given our goal to support a worldwide system, however, achieving these goals is not always possible. To provide clients with local access to the location service throughout the network, the location service must be a *distributed* service itself, consisting of multiple location servers located everywhere in the network.

Idempotency

We desire the operations in the location service to be idempotent. The result of performing an idempotent operation several times is the same as performing it once. Idempotency is a useful characteristic since it enables the location service to redo an operation, for example, if it forgot whether it already performed the operation due to a server crash. The location service therefore does not need to go to great lengths (i.e., use expensive resources) to guarantee that it will remember it performed the operation in the first place. For instance, the fact that an object in our model has a *set* of contact address already simplifies the implementation of the update operations since operations on a (mathematical) set are idempotent by nature.

Independence

We also desire that the location data is organized into a collection of relatively small, simple, mutually independent, and self-contained records. In that case, a lookup or update operation will always have to deal with only a single record. This principle has two related consequences. First, these self-contained records improve efficiency by simplifying the partitioning of the workload load (i.e., the set of records) and distributing it over multiple machines. Second, these self-contained records simplify the isolation of concurrent operations within one machine since we do not have to worry about atomically updating complex data structures.

Simplicity

Since we do not know what the actual characteristics (e.g., the average number of replicas or the mobility patterns) of our objects will be, we do not yet know how to optimize our location service, apart from general optimizations such as the exploitations of locality. We therefore do not focus on optimizations for specific situations, and support only the most basic operations needed (i.e., our lookup and update operations) and consider only the most basic replication and mobility patterns.

Best-effort Guarantees

The location service provides only minimal guarantees on its operations, that is, it provides a guarantee only when needed, otherwise the location service just does its best to get a (good) result. For instance, the lookup operation might return the contact address of a nearby replica, instead of the *nearest* replica. Guaranteeing the nearest replica would require much more resources, and possibly take longer to complete. The location service does, for instance, guarantee that when a delete operation has finished (successfully) that the deleted contact address can no longer be found since this guarantee results in a performance increase for the lookup operation.

Modularity

The location service has to support various requirements. To ease the implementation of the location service, we want to separate these requirements, and deal with them independently. We therefore desire a modular design with specific modules or layers that deal with specific problems and requirements, and that interact through small, simple, and well-defined interfaces.

2.3 Basic Design

In this section we describe the basic design of the Globe location service, and present the problems that are dealt with in the following chapters.

2.3.1 Distributed Search Tree

To exploit locality in our location service, we partition the underlying wide-area network into a **hierarchy of domains**. The domains cover regions of the network chosen on geographical, network-topological, and administrative boundaries. For instance, a domain at the lowest level, called a **leaf domain**, might cover a city, while the next level up might cover a state or country. The highest-level domain, called the **root domain**, covers the whole network.

We associate a **directory node** with every domain in the hierarchy. The directory node is responsible for storing location information, such as contact addresses, of the objects in its domain. The directory nodes together form a **distributed search tree**. All lookup and update operations are initiated at the leaf nodes of the search tree, which provide clients with local access to the location service.

The distributed search tree represents a **distance metric** on the underlying network. The lookup and update operations use this metric to improve the locality of the way they search and store contact addresses. The central notion is that all contact addresses located within a domain can be considered to be equally far away from a client located in that same domain, and that by moving up the search tree the domains get bigger and the distances larger. To be efficient, the lookup and update operations should operate in the smallest domain possible.

Figure 2.1 shows, as an example, a part of a domain hierarchy and its distributed search tree. The example shows six European cities as leaf domains. The leaf domains are contained in three country domains. The country domains are, in turn, contained in the European domain, which is part of the World domain. A contact address in Paris is therefore located simultaneously in the Parisian, French, European, and World domains.

The structure of the search tree of Figure 2.1 is determined solely by geography. The choice of using only geography in this figure is based on didactical grounds. We could also have depicted a search tree based only on network topology, with a root domain covering the whole Internet, divided into child domains for all autonomous systems (AS), and the AS domains divided into domains for the areas inside the AS domains. To simplify our discussions and examples, however, we continue using search trees based on geography throughout this dissertation since the geographical relations between the various continents, countries, and cities can be considered common knowledge.

A directory node maintains a separate **contact record** to store the location information (e.g., contact addresses) for each object in its domain. The contact record is divided into a set of **contact fields**, one for every child node of the directory node. A contact field contains the location information of the object related to the field's child domain. In the case of a leaf domain, a contact record has one contact field that contains location information for the whole leaf domain. A contact field is either empty, stores one or more contact addresses, or stores a single **forwarding pointer**. A contact record is said to be **empty** if all of its contact fields are empty. A directory node discards a contact record once it becomes empty.

The presence of a forwarding pointer in a contact field indicates that there are contact

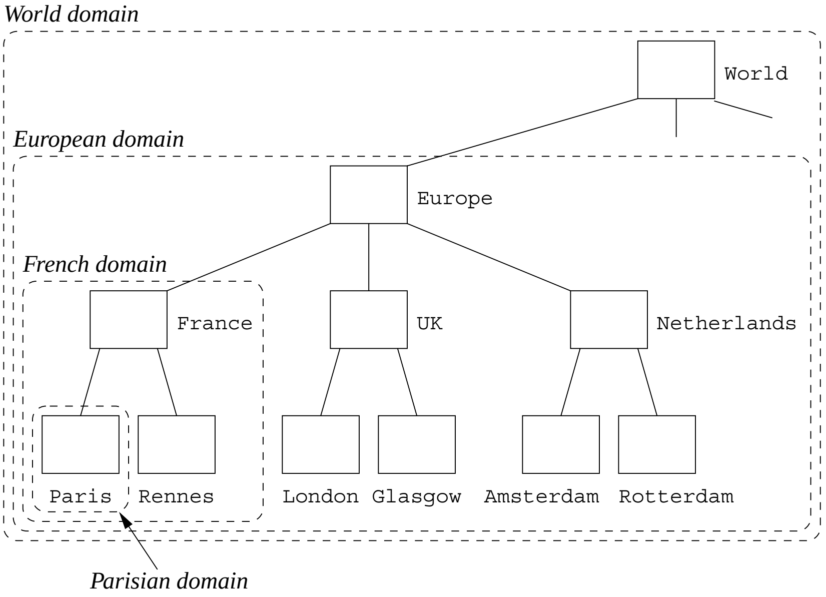


Figure 2.1: Example of a domain hierarchy with its associated search tree.

addresses located in the associated child domain. The forwarding pointers of an object together form a collection of **paths of forwarding pointers** in the search tree, each path starting at the root node and ending in a directory node that actually stores a contact address. Since there is such a path for every contact address, an address can always be found by following its path starting at the root node.

Figure 2.2 shows an example of a contact record from the `Europe` node. The contact record consists of three contact fields: the left contact field for the French domain, the middle contact field for the UK domain, and the right contact field for the Netherlands domain. The French contact field contains the contact addresses of three replicas located in France. The UK contact field contains a forwarding pointer, indicating that contact addresses can be found in the UK domain, either at the `UK` node or one of its children. The Netherlands contact field is empty, meaning that the object does not have replicas in the Netherlands.

Following our design principles, we ensure that all changes made to a contact record by an update operation are idempotent. We distinguish four types of changes:

- Adding a contact address
- Removing a contact address
- Adding a forwarding pointer

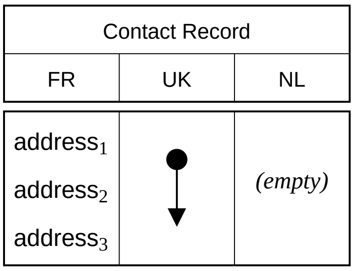


Figure 2.2: Example of a contact record from the `Europe` node.

- Removing a forwarding pointer

All four changes modify only a single contact field. The contact field behaves like a (mathematical) set when it comes to storing contact addresses, neither adding the same contact address a second time nor removing the same contact address a second time changes the contact field. The same applies to a forwarding pointer, adding or removing a forwarding pointer a second time does not change the contact field.

Figure 2.3 shows our example search tree with the contact records of a single object. Since the contact address sets of different objects are independent, the different contact records stored by a directory node are also independent. To simplify our figures, we therefore always show a search tree with the contact records of only a single object.

In the figure, the object consists of two replicas, one located in Paris and the other in Glasgow. While the contact address of the replica in Paris is stored directly at the `Paris` leaf node, the address of the replica in Glasgow is actually stored one level higher, at the `UK` node. A contact address is usually stored in the directory node of the leaf domain in which it resides, but it can also be stored in a directory node higher in the tree. Storing contact addresses higher in the tree can improve the efficiency of the location service when the associated object is mobile, as we discuss in Chapter 4. Other nodes, like the `Netherlands` node, do not contain a contact record for this object since such a contact record would be empty.

In our example, both addresses can be found by following a path of forwarding pointers. For the contact address in the Parisian domain this path consists of the `World`, `Europe`, `France`, and `Paris` nodes. For the contact address in the Glasgow domain this path consists of the `World`, `Europe`, and `UK` nodes. Notice that the two paths of forwarding pointers share the forwarding pointer from the `World` node to the `Europe` node.

Consider what would happen in our example when a client in Rennes would be interested in using the object. The client would go to the directory node of its leaf domain to contact the location service, and initiate a lookup operation. Given our locality goal, the client would like the lookup operation to return the contact address in the Parisian domain. Furthermore, we would like the lookup operation to use only resources in the

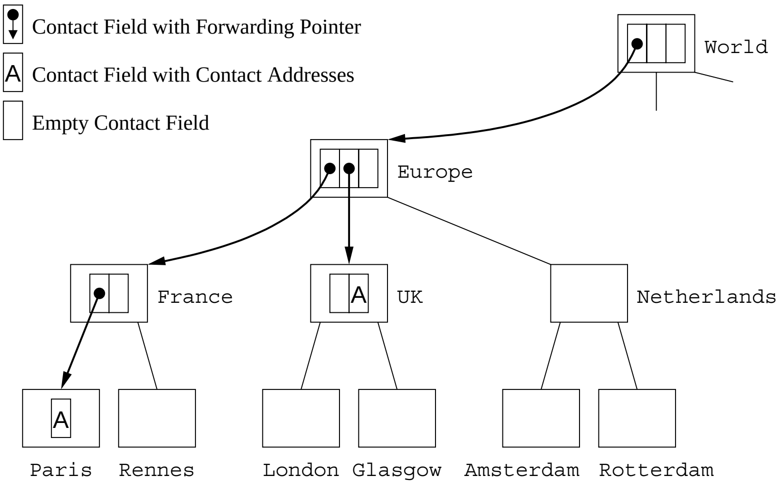


Figure 2.3: Example of a search tree with the contact records of a single object.

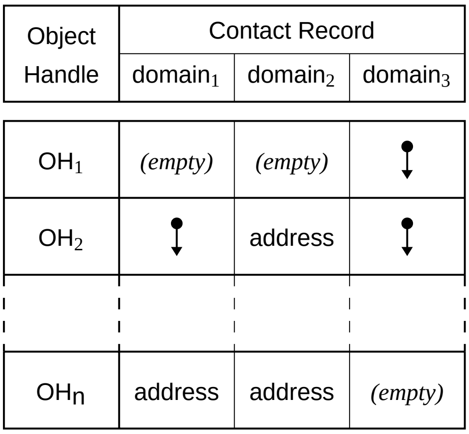
French domain (i.e., the `France`, `Paris`, and `Rennes` nodes). The method to achieve our locality goal is described later in this chapter, in Section 2.5.

Since a directory node has to store a contact record for every object in its domain, it has to maintain a **contact record database**. Figure 2.4 shows a typical contact record database. The contact record database is simply a persistent table indexed by the object handles that identify the objects. All contact records in the database have the same number of fields because this depends on the number of children of the directory node.

The database is queried and modified by lookup and update operations. Since there is no (consistency) relation between the contact records stored in the database, the database has to support the reading and writing of single contact records only. It does not need to support complicated queries or atomic group modifications on multiple records. Furthermore, since the contact records are independent, the reading and writing of contact records can also be handled concurrently.

A directory node associates a **lease** [Gray and Cheriton, 1989; Duvvuri et al., 2000] with every contact address stored in its contact record database. When this lease runs out, the node automatically initiates a *deleteAddress* operation for the associated address. This operation removes the contact address from the contact record database, and will also remove the path of forwarding pointers to the contact address from the search tree, if needed. A client is thus forced to regularly extend the contact address leases to ensure the addresses remain stored. We use these contact address leases to enable the location service to avoid storing a large number of invalid or useless object handles and contact addresses, which clients have forgotten to remove. The lease system is also used for fault tolerance and tree management purposes, as discussed in Chapters 5 and 7, respectively.

Comparing Figure 2.5 with Figure 2.3 shows the effects of an insert operation on



Object Handle	Contact Record		
	domain$_1$	domain$_2$	domain$_3$
OH$_1$	*(empty)*	*(empty)*	↓
OH$_2$	↓	address	↓
OH$_n$	address	address	*(empty)*

Figure 2.4: Example of the persistent contact record database maintained by a directory node. The database contains the contact records of *n* objects. The database is indexed by object handle only.

the search tree. Our example object has a new replica in Amsterdam, and a contact address has therefore been inserted at the `Amsterdam` leaf node. The result of the insert operation is the creation of a new contact record with the new contact address at the `Amsterdam` leaf node, the creation of a new contact record with a forwarding pointer at the `Netherlands` node, and the addition of a forwarding pointer at the `Europe` node.

Comparing Figure 2.6 with Figure 2.5 shows the effect of a delete operation on the search tree. The replica in Glasgow of our example object has disappeared, and its contact address has therefore been deleted at the `Glasgow` leaf node. The result of the delete operation is the removal of the contact record storing the contact address at the `UK` node, and the removal of a forwarding pointer at the `Europe` node.

2.3.2 Invariants on the Search Tree

To implement the lookup and update operations correctly, we need to know precisely what a valid search tree is. For instance, we need to know what an operation can expect from the rest of the search tree when it finds a contact address or forwarding pointer in a node. We use the following three conditions to specify the structure of a valid search tree. The conditions must be true when there are no changes taking place in the search tree (i.e., no running update operations) and no failures have occurred. Chapter 5 examines the structure of the search tree in the presence of failures. In the three conditions (and in the rest of this chapter), we use the notation $dir(D)$ to denote the directory node of a domain *D*, and the notation $dom(N)$ to denote the domain of a directory node *N*.

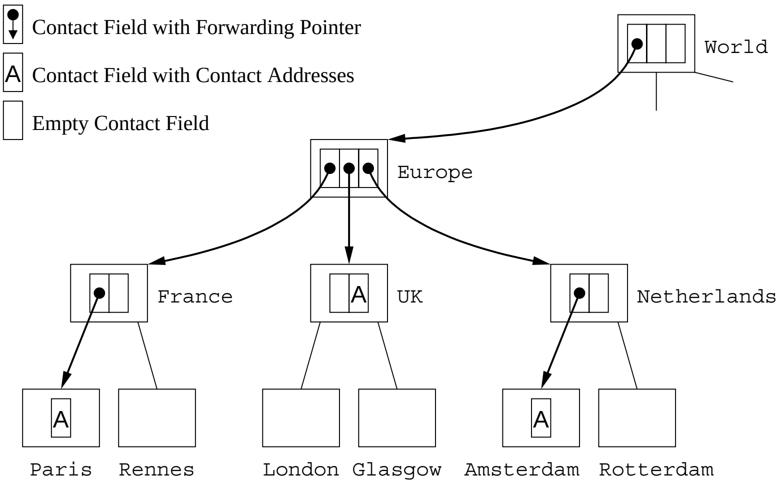


Figure 2.5: The search tree example of Figure 2.3 with a new contact address from the Amsterdam leaf domain inserted.

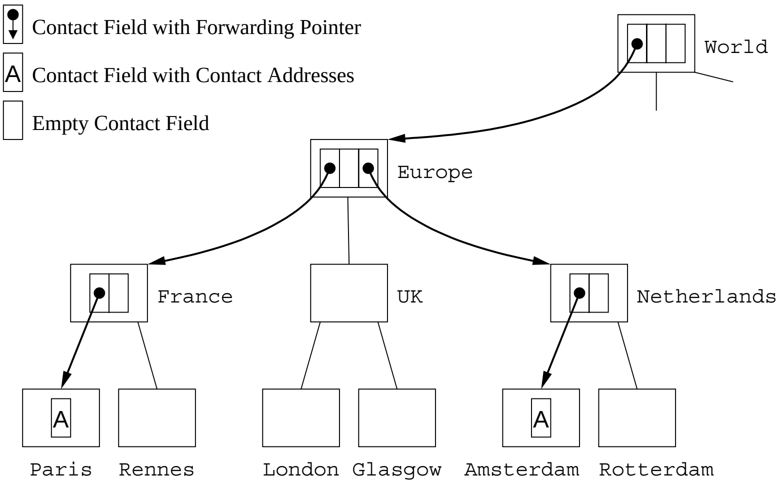


Figure 2.6: The search tree example of Figure 2.5 with the contact address from the Glasgow leaf domain deleted.

C1 *A contact address from a leaf domain D is stored at dir(D), or at the directory node of an enclosing (higher-level) domain of D.*
This condition implies that a contact address from leaf domain D can be stored only at a directory node that lies on the path from leaf node $dir(D)$ up to the root node.

C2 *For each node N, the contact record for object O at node N stores a forwarding pointer to a child node of N if and only if the contact record for O at that child is not empty.*
This condition states that we do not accept dangling forwarding pointers in our search tree. This condition has two important consequences. First, if we follow a path of forwarding pointers, we will eventually reach a contact record containing one or more addresses. Second, if a node has a nonempty contact record, there exists a path of forwarding pointers from the root down to this node.

C3 *A contact field can contain either a forwarding pointer or a set of contact addresses, but not both.*
Together with the previous conditions, this condition implies that as soon as we encounter a contact field containing addresses, we can be sure that we have found all contact addresses that lie in the child domain represented by that contact field.

When these three conditions are met, the search tree is said to be **globally consistent**. An update operation needs to ensure it leaves the search tree in a globally consistent state.

2.3.3 Specific Problems

In the current design, the root of the distributed search tree is the source of an obvious scalability problem. Since every object has paths of forwarding pointer from the root node down to every node that stores a contact addresses, all known objects have a contact record stored at the root. Given that we want to support 10^{12} objects worldwide, the root node has to store and process lookup and update requests for 10^{12} contact records. This load is clearly too much to handle for a single physical machine. Chapter 3 describes a way to use the independence design principle to extend the current design, and solve this problem.

At any time, there will be a huge number of lookup and update operations in progress in the search tree. To ensure efficient use of its resources, the location service must handle these operations concurrently where possible. However, concurrency should be used carefully, or otherwise there is the risk of inconsistent data structures due to race conditions. Fortunately, we can handle most operations concurrently with little effort since the contact records of different objects are independent and no inconsistencies can arise. However, we still have to deal with concurrent operations on the same object, especially concurrent update operations.

The usual way to avoid race conditions is to prevent concurrency temporarily using locking mechanisms, allowing only one operation to access a data structure at a time. Unfortunately, this method should be used only on small data structures since it would otherwise prevent concurrency in the tree. In other words, we do not want to use a locking scheme where parts of the search tree are temporarily unavailable. Luckily, contact records from different object handles can be locked independently, given the independence principle. We have increased concurrency further by devising a method to handle update operations on the same object concurrently. We describe this method in Chapter 5.

Since we are dealing with a large number of resources, such as networks, hosts, storage, we expect faults to occur regularly in the location service. However, since the location service plays a pivotal role in the way clients access objects, we need to ensure it remains available as much as possible. Furthermore, failures are not allowed to lead to inconsistencies in the search tree. Ensuring consistency in the face of failures is especially difficult when dealing with concurrent update operations. In Chapter 5, we describe our method to make the location service fault tolerant and ensure availability. The method is based on the idempotency design principle.

Since the location service plays an important role in the process of contacting an object, it is also an important target for malicious clients wanting to compromise this process. We therefore want to protect the mapping from object handle to set of contact addresses from tampering. Without protection any client can remove valid contact addresses and insert phony ones. Providing access control to contact records is, however, not enough. Even with the protection of the update operations, we still have to protect the communication between directory nodes, and between leaf nodes and clients, to ensure that messages are not altered. For instance, without protection, a malicious client can alter a location service reply to a lookup request, introducing incorrect results. These security problems and their solutions are dealt with in Chapter 6.

The distributed search tree is not static. We expect that the search tree will be restructured regularly to continue to provide locality in a changing (network) environment. Examples of these changes are the addition and removal of leaf nodes and the splitting and merging of domains. To maintain consistency, these changes require that contact addresses and forwarding pointers will be moved and copied between tree nodes. Furthermore, since we cannot simply stop all the lookup and update operations in the location service, these search tree changes have to be made concurrently with normal tree operations. These, what we call, tree management methods are discussed in Chapter 7.

2.4 Search Tree Operations

In the following three sections we describe simplified versions of the lookup and update operations. These versions are meant to explain only the overall structure of the operations. We ignore issues like concurrency, fault tolerance, and security, and focus on how the current design exploits locality. Before diving into the details, however, we first introduce the communication mechanism and the notation used.

2.4.1 Communication in the Search Tree

The search tree is implemented as a set of independently running processes, each implementing a single directory node. These processes execute **procedures** that implement the lookup and update operations. The processes communicate with each other using **remote procedure calls** (RPCs) [Birrell and Nelson, 1984]. In the RPC model, a directory node (the caller) can start the execution of a procedure at an other node (the callee).

To start the execution of a procedure, the caller sends a message with an RPC request to the callee. When the callee receives this RPC request, it will start executing the specified procedure. The execution of a procedure can result in the callee starting a remote procedure in a third directory node. When the callee has finished executing the procedure, it sends back to the caller a message with the RPC reply containing a return value, for instance, the contact addresses found during a lookup operation. We examine the failure semantics of the RPC system in Chapter 5, and assume for now reliable communication and nodes that do not crash.

We use **node identifiers** in our algorithms to refer to directory nodes in the search tree. For instance, every directory node records the node identifier of its parent and child nodes. Node identifiers are universally unique identifiers that can be passed around freely within the location service. We always add the node identifier of the sending node to the parameter list of the RPC request. The receiving node uses this parameter, for instance, during a lookup operation to determine how to continue traversing the search tree.

A client starts a lookup or update operation in the location service by sending an RPC request to a leaf node, usually the leaf node of the domain in which the client resides. The leaf node needs to handle this RPC request slightly differently since a client does not have a node identifier. Since we do not want to make the distinction between the caller being a client or another directory node in our lookup or update procedures, we let the RPC system deal with this problem. When an RPC request comes from a client, the RPC system at the leaf node uses the leaf node's own node identifier in the parameter list. The procedure uses this node identifier as if it were the identifier of another directory node, unaware it is its own node identifier and that the request actually came from a client.

2.4.2 Notation

We specify our procedures in high-level Ada-like pseudocode. This allows us to focus on the structure of the operations, instead of getting distracted by implementation details. A directory node invokes a procedure in another node by using an explicit RPC statement. An RPC statement is expressed by the following construct.

```
call invocation at nodeid;
```

Where *invocation* consists of the name of the procedure to be executed and the actual parameters to be used. The node identifier *nodeid* specifies at which node to execute the operation. The following is an example of an RPC statement.

```
(1) type ObjectHandle is opaque;
(2) type ContactAddress is opaque;
(3) type NodeID is opaque;
(4) type ContactField is
(5)     record
(6)         addrSet : set of ContactAddress := $\emptyset$;          -- Addresses in subdomain
(7)         isPtr : Boolean := false;                         -- Forwarding pointer to child node
(8)     end record;
(9) type ContactRecord is set (NodeID) of ContactField := $\emptyset$;
```

Listing 2.2: Data types used in the example code.

```
call insertAddress( oh, addr, thisNode ) at parent;
```

In this example, a node invokes the procedure *insertAddress* at its parent node. It provides an object handle as the first, a contact address as the second, and its node identifier as the third parameter.

Listing 2.2 shows the main data types used in our procedures. The *ObjectHandle*, *ContactAddress*, and *NodeID* type (lines 1–3) are opaque. Values of an opaque data type can be compared for equality and used as an index value. The *ObjectHandle* type is further defined in Chapter 3. The *NodeID* type is also used as a domain identifier since there is a 1-to-1 mapping between directory nodes and domains. The *ContactRecord* and its constituent the *ContactField* are defined in lines 4–9. A *ContactField* itself consists of two fields: *addrSet*, a set of contact addresses (line 6); and *isPtr*, a boolean value indicating the presence of a forwarding pointer (line 7). Both fields have a defined initial value: *addrSet* is given the empty set as an initial value, and *isPtr* is initialized to *false*.

Line 9 defines a *ContactRecord* as an indexed set of contact fields. An indexed set provides a mapping from keys to values. In the case of a contact record, the node identifier is the index type that maps to a contact field. The node identifier identifies the child domain associated with the contact field. At a leaf node, a contact record consists of a single contact field. This contact field is associated with the node identifier of the leaf node itself. This practice is consistent with the standard node identifier parameter in an RPC request sent by a client, as described above.

Listing 2.3 shows the variables used by the procedures. The variables are global with respect to the directory node in which they are used. Line 1 declares the contact record database *crDatabase*. This global variable is an indexed set of contact records that uses the object handle as index type. The object handle identifies the object to which the contact record belongs. If an unknown object handle is used to index the database, the indexing operation returns the *NIL* value. The fact that the contact record database is persistent is not visible in this declaration. Line 2 declares the node identifier *thisNode* that stores the identifier of the local directory node. Line 3 declares the node identifier *parent* that identifies the parent of this directory node. Line 4 declares *children*, a set of node identifiers that identify the children of this directory node.

```
crDatabase : set (ObjectHandle) of ContactRecord;
thisNode : NodeID;
parent : NodeID;
children : set of NodeID;
```

Listing 2.3: Global variables used in the example.

2.5 Simplified Lookup Operation

The goal of the simplified lookup operation is to provide a client with a single contact address from the set of contact addresses associated with the specified object. The client specifies the object using its object handle. The lookup operation should retrieve the contact address of a replica near the client. To lessen the impact of the operation on the location service, it should preferably use only resources located near the client requesting the lookup operation. The client initiates the lookup operation at the leaf node of the domain in which it resides.

2.5.1 General Structure

The simplified lookup operation can, in principle, be divided into two parts. In the first part, the lookup operation checks for the presence of a contact record of the specified object, while traveling from the leaf node toward the root. When a contact record is found, the second part starts. In the second part, the lookup operation follows the path of forwarding pointers, starting in the contact record, downward to a contact record that stores a contact address.

Figure 2.7 shows the execution of a simplified lookup operation in our running example. A client located in Amsterdam wants to use the object and starts a lookup operation by sending a lookup request to the `Amsterdam` node, in step 1. In response, the `Amsterdam` node checks whether it knows the object. Since it does not know the object, the `Amsterdam` node sends, in turn, a lookup request to the `Netherlands` node, in step 2. This time the `Netherlands` node checks whether it knows the object, and the object is again unknown. The `Netherlands` node therefore sends a lookup request to the `Europe` node, in step 3. The object is known in the European domain.

The `Europe` node has associated with the object a contact record with two forwarding pointers. The node therefore needs to decide which forwarding pointer to follow. Since the both forwarding pointers can be considered equivalent (both point indirectly to contact addresses in the European domain), the lookup operation can follow either one. In our example, the `Europe` node follows the UK forwarding pointer by sending a lookup request to the `UK` node, in step 4. This node finally finds a contact address, and the lookup operation returns with this address, in step 5–7, along the reverse path it traveled. The contact address is returned to the client in step 8.

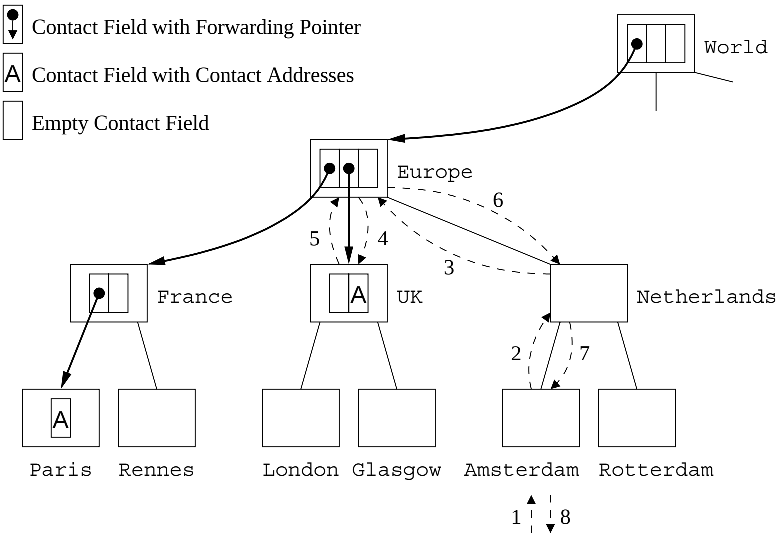


Figure 2.7: The execution of the simplified lookup operation in our example search tree. The operation is initiated by a client in Amsterdam.

2.5.2 Optimizing for Locality

As mentioned in the previous section, the distributed search tree represents a distance metric on the underlying network. The lookup operation uses this distance metric to improve the locality of the way it searches for contact addresses in the following way. The lookup operation starts searching for the object in a leaf domain, and by trying the smaller (lower level) domains first, the lookup operation effectively searches in the nearby regions before trying regions located farther away. If no contact address is found in a domain, the lookup operation extends its search in progressively larger enclosing domains until an address is found. This method of searching finds nearby contact addresses first, and uses only nearby resources when possible. This use of locality is similar to the use of expanding ring searches in multicast networks.

Unfortunately, when there is no locality (i.e., the client and contact address are located far apart), the simplified lookup operation will need to visit the root node to find the path of forwarding pointers, and follow the pointers down toward a contact address. What is even worse is that when the client sends the same lookup request a second time, the simplified lookup operation will follow the same path in the search tree. We want to avoid this kind of tree traversal when possible since it uses a large amount of resources throughout our system. In Chapter 4, we describe a caching technique that avoids unnecessary tree traversals.

```
(1)  procedure lookupAddress(oh : ObjectHandle;
(2)                          caller : NodeID) returns ContactAddress is
(3)    addr : ContactAddress;                        −− contact address found
(4)    cr : ContactRecord;                           −− contact record found
(5)  begin
(6)    −− Retrieve a copy of the contact record from the database
(7)    cr := crDatabase(oh);
(8)    if $cr \neq NIL$ then
(9)      −− The contact record contains addresses or forwarding pointers
(10)     choose any child with $cr(child).addrSet \neq \emptyset$;
(11)     if $child \neq NIL$ then
(12)       −− The contact record contains addresses
(13)       choose any addr with $addr \in cr(child).addrSet$;
(14)       return addr;
(15)     end if;
(16)     −− Apparently the contact record contains only forwarding pointers
(17)     choose any child with cr(child).isPtr;
(18)     return call lookupAddress(oh, thisNode) at child;
(19)   elsif $parent \neq NIL$ then
(20)     −− The object is unknown at this node, try the parent
(21)     return call lookupAddress(oh, thisNode) at parent;
(22)   else
(23)     return NIL;                      −− The object is even unknown at the root
(24)   end if;
(25) end lookupAddress;
```

Listing 2.4: The simplified *lookupAddress* procedure.

2.5.3 Implementation

Listing 2.4 shows the procedure *lookupAddress*, used to implement the simplified lookup operation. The procedure takes two parameters, the object handle *oh* for which the client wants a contact address, and the standard parameter *caller*, specifying the directory node that sent the *lookupAddress* request, as shown in lines 1–2. The *lookupAddress* procedure returns a single contact address. It uses two local variables, as shown in lines 3–4. Contact address *addr* stores the contact address found, and contact record *cr* temporarily stores the object's contact record.

The *lookupAddress* procedure starts by retrieving the locally stored contact record of the object, in line 7. If the contact record exists, the object is known in this domain. The procedure uses the location information stored in the contact record to retrieve a contact address, in lines 9–18.

The procedure first checks if the contact record has one or more contact addresses, in lines 10–11. If the record has addresses, the procedure picks one at random, in line 13, and returns it, in line 14. Otherwise, the contact record apparently contains only forwarding pointers. The *lookupAddress* procedure picks a forwarding pointer at random, in line 17, and uses it to retrieve a contact address at the child node associated with that pointer, in line 18. Given a globally consistent search tree, this should always succeed.

If the object is unknown at this directory node, no contact record was found in line 7. In this case, the lookup operation should continue at the parent node. If the parent node exists, the procedure sends a *lookupAddress* request to the parent and awaits its reply, in lines 19–21. If no parent node exists, the lookup operation has arrived at the root without finding the object. To indicate that the object is unknown, the *NIL* value is returned, in line 23.

2.6 Simplified Insert Operation

The goal of the insert operation is to add a new contact address to the set of contact addresses associated with a specified object. The client specifies the object using its object handle. A contact address always belongs to a specific leaf domain, and it needs to be stored in one (and only one) of the directory nodes located on the path from the associated leaf node to the root node. To ensure the contact address can be found, the insert operation also needs to add a path of forwarding pointers from the root node down to the node that stores the contact address. The insert operation should also preferably use only nearby resources to lessen its impact on the location service.

2.6.1 General Structure

The insert operation uses a **distributed decision-making process** to decide at which directory node to store the contact address. This process (potentially) involves all nodes from the leaf node up to the root node. In this distributed process, every node first makes a preliminary decision whether it wants (or needs) to store the contact address itself, and based on this decision tentatively modifies its contact record. The decision depends on the mobility pattern of the object, as further explained in Chapter 4. The node then asks its parent node for approval of the modification. The parent node can override the node's decision and decide to store the contact address itself, in which case the node has to undo its tentative modification. The parent can also agree with the node's preliminary decision, and the node's tentative modification becomes permanent. The execution of an insert operation can therefore be divided in an upward and a downward half. In the upward half of the insert operation, the nodes decide tentatively to store the contact address or a forwarding pointer; and in the downward half the modifications are undone or made permanent.

Figure 2.8 shows the start of the upward half of an insert operation in our running example. A client located in Amsterdam wants to insert a new contact address into the set of contact addresses associated with the object. The client starts the insert operation by sending an insert request to the `Amsterdam` leaf node, in step 1. The `Amsterdam` leaf node decides (by default) that it wants to store the new contact address. Since the object was previously unknown, a new contact record has to be made to tentatively store the contact address. Tentative data is shaded gray in our example figures.

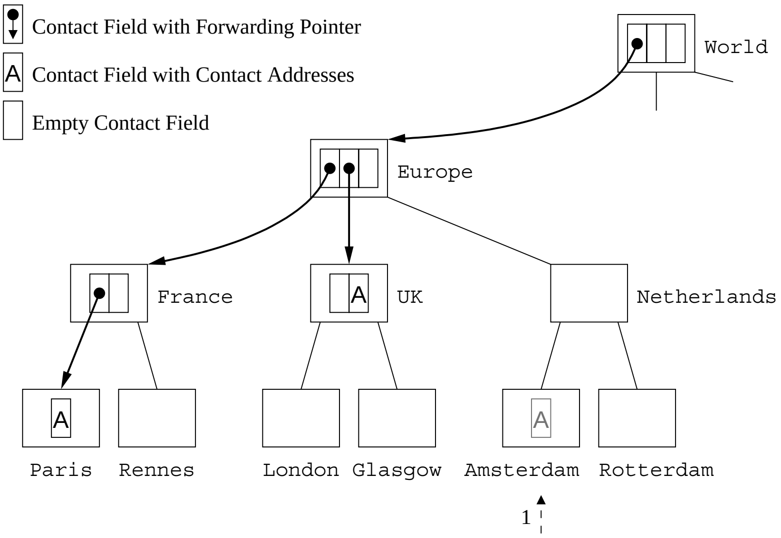


Figure 2.8: In step 1 of the execution of the simplified insert operation (initiated at the Amsterdam leaf node), a contact address is inserted at the Amsterdam leaf node.

Since the object was previously unknown, the Amsterdam node sends an insert request to the Netherlands node to obtain that node's approval for the tentative change, as shown in step 2 in Figure 2.9. The Netherlands node approves the request and tentatively stores a forwarding pointer. To store the forwarding pointer, a new contact record is also created at this node.

Since the object was also unknown at the Netherlands node, this node asks approval for its tentative change from the Europe node, as shown in step 3 in Figure 2.10. The Europe node agrees with the Netherlands node, and stores a forwarding pointer to that node. Since the object was already known at the Europe node, there exists a path of forwarding pointers to the Europe node (conform consistency rule C2), and the upward half of the insert operation stops here.

Figure 2.11 shows the downward half of the insert operation. Since the object was already known at the Europe node, its tentatively inserted forwarding pointer becomes permanent immediately. The Europe node then sends its approval back to the Netherlands node, in step 4. Based on this approval, the Netherlands node knows its forwarding pointer is permanent as well, and sends back its approval to the Amsterdam node, in step 5. The Amsterdam node then knows the contact address is permanently stored, and, in step 6, informs the client that the insert operation was successful.

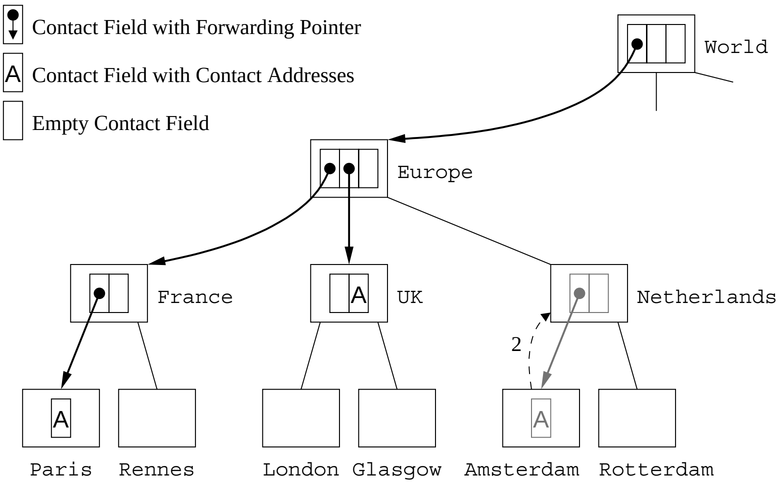


Figure 2.9: In step 2 of the execution of the simplified insert operation (initiated at the `Amsterdam` leaf node), a forwarding pointer is inserted at the `Netherlands` node.

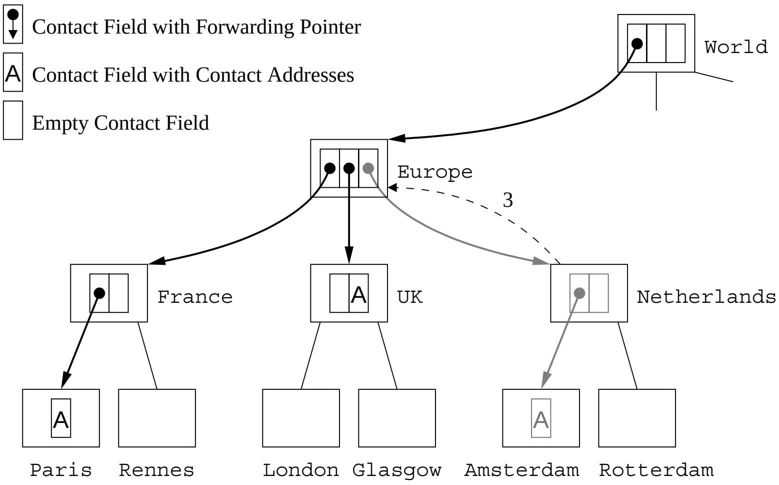


Figure 2.10: In step 3 of the execution of the simplified insert operation (initiated at the `Amsterdam` leaf node), a forwarding pointer is inserted at the `Europe` node.

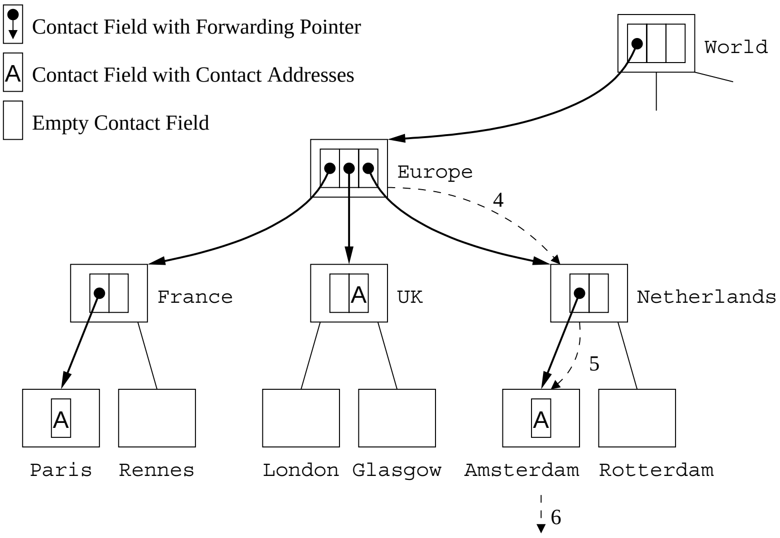


Figure 2.11: In the second half, step 4–6, of the execution of the simplified insert operation (initiated at the `Amsterdam` leaf node), the nodes send their RPC replies.

2.6.2 Design Principles Used

The insert operation also uses the locality design principle to improve scalability. When an object is already known in domain D (the European domain in our example), there is a path of forwarding pointers established from the root down to domain D. The insert operation therefore does not need to travel further up to the tree than directory node $dir(D)$, and the resources used by the insert operation are confined to domain D. This property is the result of the upper part of the path of forwarding pointers being shared.

Given our design principle of using a best-effort approach, we cannot guarantee when the location service completes the insert operation. We therefore do the next best thing, which is making the result of the insert operation (i.e., the contact address) visible in the search tree as soon as possible. We do this by tentatively inserting the contact address or forwarding pointer at a node before the node gets an approval from its parent. This enables concurrent lookup operations to find the contact address. We can safely insert the contact address or forwarding pointer since the answer from the parent only determines *where* in the tree the contact address is eventually stored; a client is interested only in the contact address itself.

2.6.3 Implementation

The implementation of the simplified insert operation is divided over two procedures, *insertAddress* shown in Listing 2.5, and *insertPointer* shown in Listing 2.6, each with its own RPC request. We use two separate requests (and procedures) because a directory node can make two different types of request. An *insertAddress* request indicates that a child node wants its parent to store a contact address, and an *insertPointer* request indicates that a child node wants its parent to store a forwarding pointer, implicitly asking the parent's *approval* for storing an address. This division is also present in more advanced versions of the insert operation that we discuss in later chapters. A client initiates an insert operation by sending an *insertAddress* request to a leaf node.

The insertAddress Procedure

The *insertAddress* procedure takes three parameters, as shown in lines 1–3 of Listing 2.5. Object handle *oh* identifies the object of which the contact address set should be modified. Contact address *addr* specifies the address that should be inserted. The standard parameter *caller* specifies the directory node that sent the *insertAddress* request. This parameter specifies a child node of the receiving node if the procedure executes at an intermediate node, or the receiving node itself if the procedure executes at a leaf node (i.e., the request came from a client).

The *insertAddress* procedure returns a boolean value that tells the caller whether it is supposed to store the contact address or not. The simplified version of the *insertAddress* procedure always returns the boolean value *false*, as seen in line 33. The reason for this is that when a caller sends an *insertAddress* request, the contact address is always stored at the receiving node, or higher in the tree. The caller therefore never needs to store the address itself. In later versions of the *insertAddress*, the procedure can return *true* as well.

The *insertAddress* procedure uses two variables, as shown in lines 4–5. The contact record variable *origCR* stores the original contents of the contact record in the contact record database. The boolean variable *approved* stores the RPC reply from the parent node. This reply tells whether this node is allowed to store the contact address or not.

The *insertAddress* procedure starts by retrieving a copy of the contact record from the database using the object handle, in line 8. If the object is unknown in this directory node, there is no contact record in the contact record database (line 9), and a new contact record needs to be created and initialized, in lines 10–11. The procedure *initializeCR* initializes the contact record by inserting empty contact fields for all the children. If the procedure executes at a leaf node (indicated by the *children* parameter set to the empty set), one empty contact field is inserted using *thisNode* as node identifier. The procedure then (tentatively) adds the contact address to the contact record in the database, in line 14. The original contact record *origCR* is left unchanged, so we can use it later to undo the change.

To determine whether approval is needed from the parent, the procedure checks, in line 16, whether the contact record was originally empty (i.e., did not store contact addresses or forwarding pointers) and a parent node exists. If contact with the parent is needed, the procedure needs to determine its own desired final state: storing the contact address or not. This is determined by the *doStoreHere* function

The *doStoreHere* function returns *true* when the node wants to store the contact ad-

```
procedure insertAddress(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) returns Boolean is
  origCR : ContactRecord;                          -- original contact record
  approved : Boolean;                              -- parent approves address insert
begin
  -- Retrieve a copy of the contact record, or create one, if needed
  origCR := crDatabase(oh);
  if origCR = NIL then
    origCR := new ContactRecord;
    initializeCR(origCR, children, thisNode);
  end if;
  -- Add the address (possibly temporarily) to the record.
  crDatabase(oh)(caller).addrSet := origCR(caller).addrSet + {addr};
  -- Ask the parent for approval, if needed
  if isEmpty(origCR) and parent $\neq$ NIL then
    -- Determine our own desired action, and ask the parent for approval
    if doStoreHere(origCR)                         -- See Chapter 4
      then approved := call insertPointer(oh, addr, thisNode) at parent;
       else approved := call insertAddress(oh, addr, thisNode) at parent;
    end if;
  else
    approved := true;                              -- No approval needed
  end if;
  -- Undo the local modification, if the parent does not approve
  if not approved then
    if isEmpty(origCR)
      then delete crDatabase(oh);                  -- Remove temporary record
       else crDatabase(oh) := origCR;              -- Restore original record
    end if;
  end if;
  -- The contact address is stored here, or higher in the tree
  return false;
end insertAddress;
```

Listing 2.5: The simplified *insertAddress* procedure.

dress, and *false* otherwise. We describe the design of the *doStoreHere* function in Chapter 4. Here we can simply assume that the function returns by default *true* in leaf nodes and *false* everywhere else, resulting in contact addresses being inserted in the leaf nodes. Depending on the value of *doStoreHere*, the procedure requests the parent node for approval for storing the contact address, in line 19, or requests the parent node to store the contact address, in line 20.

If the contact record already contained information or the parent node does not exist, no approval is needed (line 23). The parent's response, stored in *approved*, ultimately determines, in line 26, whether the contact address is inserted. If the parent node did not allow this node to store the contact address, the tentative change made in line 14 is undone, in lines 27–30, either by removing the contact record, if the contact record was originally empty, in line 28, or by copying the original contact record back into the contact record database, in line 29. In all cases, the contact address is stored at this node or higher in the tree, and the caller is not allowed to store the contact address, as reported in line 33.

The insertPointer Procedure

The *insertPointer* procedure takes the same parameters at the *insertAddress* procedure, as shown in lines 1–3 in Listing 2.6. The *insertPointer* procedure also returns a boolean value, telling the caller whether it is allowed to store the contact address or not. However, in contrast to the simplified *insertAddress* procedure, *insertPointer* returns *true* or *false*. This return value tells whether the calling node is allowed to store the contact address or not.

The *insertPointer* procedure uses three variables. The contact record variable *origCR* stores the original contents of the contact record in the contact record database. The boolean variable *storedAddress* shows whether the directory node has stored the contact address or a forwarding pointer. The boolean variable *approved* stores the RPC reply from the parent node.

The *insertPointer* procedure starts by retrieving a copy of the contact record from the database using the object handle, in line 9. If the object is unknown in this directory node, there is no contact record in the contact record database (line 10) and a new contact record needs to be created and initialized, in lines 11–12. The procedure first determines, in lines 15–16, what it actually can and wants to do given the state of the contact field. If the contact field already stores contact addresses, the procedure is forced by the consistency requirements to store the new address as well. If the contact field already holds a forward pointer, the requirements forces the procedure to store a forwarding pointer. Only when the contact field is empty can the procedure decide what it wants to do, using the *doStoreHere* function, in line 15. The actual modification of the database is done in line 18 or line 20. The performed modification is recorded in *storedAddress*.

The change still needs to be approved by the parent node. This approval is needed if the original contact record was empty and the parent exists (lines 23–26). If the parent did not approve the change (line 28), it is necessary to undo the tentative changes made in lines 17–20. The node either removes the contact record, if the contact record was

```
procedure insertPointer(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) returns Boolean is
  origCR : ContactRecord;                              −− original contact record
  storedAddress : Boolean;                             −− address or forwarding pointer
  approved : Boolean;                                  −− parent approves address insert
begin
  −− Retrieve a copy of the contact record, or create one, if needed
  origCR := crDatabase(oh);
  if $origCR = NIL$ then
    origCR := new ContactRecord;
    initializeCR(origCR, children, thisNode);
  end if;
  −− Add the address or pointer (possibly temporarily) to the record
  if $origCR(caller).addrSet \neq \emptyset$ or (not origCR(caller).isPtr and
                                 doStoreHere(origCR))
   then crDatabase(oh)(caller).addrSet := $origCR(caller).addrSet + \{addr\}$;
        storedAddress := true;
   else crDatabase(oh)(caller).isPtr := true;
        storedAddress := false;
  end if;
  −− Ask the parent for approval, if needed
  if isEmpty(origCR) and $parent \neq NIL$
   then approved := call insertPointer(oh, addr, thisNode) at parent;
   else approved := true;                              −− No approval needed
  end if;
  −− Undo the local modification, if the parent does not approve
  if not approved then
    if isEmpty(origCR)
      then delete crDatabase(oh);                      −− Remove temporary record
      else crDatabase(oh) := origCR;                   −− Restore original record
    end if;
    return false;                  −− The contact address is stored higher in the tree
  end if;
  −− The caller can store the contact address if we did not
  return not storedAddress;
end insertPointer;
```

Listing 2.6: The simplified *insertPointer* procedure.

originally empty (line 30), or copies the original contact record back into the contact record database (line 31). Either way, the contact address is stored higher in the tree, and the caller is not allowed to store the contact address, in line 33. If the parent did approve the modification, the node needs only to tell the caller what it is allowed to do, in line 36. The caller does the opposite of this node, that is, it stores the contact address if and only if this directory node does not.

2.7 Simplified Delete Operation

The goal of the delete operation is to remove an existing contact address from the set of contact addresses associated with a specified object. The client specifies the object using its object handle. The contact address was originally inserted at a certain leaf node, and must be deleted at the same leaf node. The delete operation needs to find the directory node storing the contact address, delete the contact address at this node, and remove those forwarding pointers that are no longer needed. The delete operation should also preferably use only nearby resources to lessen its impact on the location service.

2.7.1 General Structure

Figure 2.12 shows the start of the delete operation. It shows a client wanting to delete an existing contact address from the Glasgow domain. To start the delete operation, the client sends a delete request to the `Glasgow` leaf node, in step 1. Since the object is unknown in this directory node, the directory node has nothing to do, and sends a delete request to the `UK` node, in step 2 in Figure 2.13. The `UK` node stores the contact address, and therefore removes the contact address. Since the contact record at the `UK` node has become empty, the node removes the contact record completely.

Since the `UK` node no longer stores a contact record, the `Europe` node needs to remove its forwarding pointer to the `UK` node. The `UK` node therefore sends a delete request, in step 3 in Figure 2.14. The `Europe` node removes the forwarding pointer upon receiving this request. The delete operation stops at the `Europe` node since its contact record has two other forwarding pointers remaining. Figure 2.15 shows the replies to the delete requests being sent, in step 4–5. The `Glasgow` node then informs the client that the delete operation was successful, in step 6.

2.7.2 Design Principles Used

The delete operation also uses locality to improve scalability. When the object being modified remains known in domain D (the European domain in our example), the path of forwarding pointers to directory node $dir(D)$ may not be deleted. The delete operation therefore does not need to travel further up the tree. This is again the result of the upper part of the path of forwarding pointers being shared.

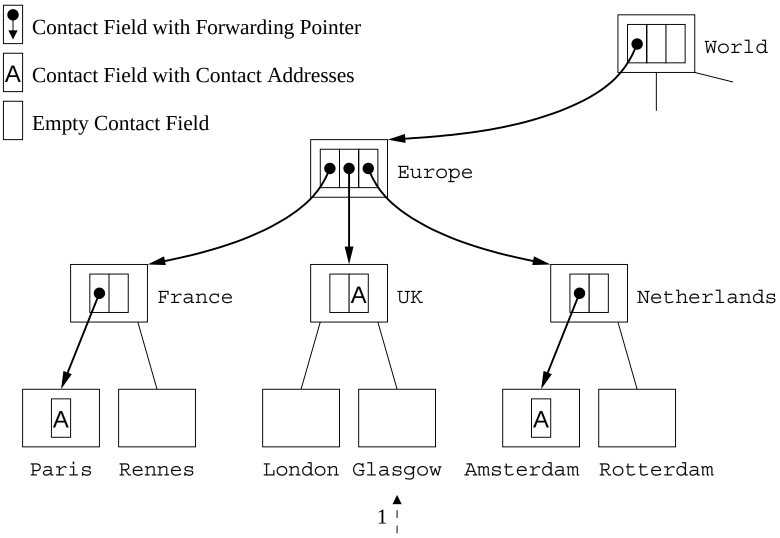


Figure 2.12: In step 1 of the execution of the simplified delete operation (initiated at the `Glasgow` leaf node), no contact record is found at the `Glasgow` node.

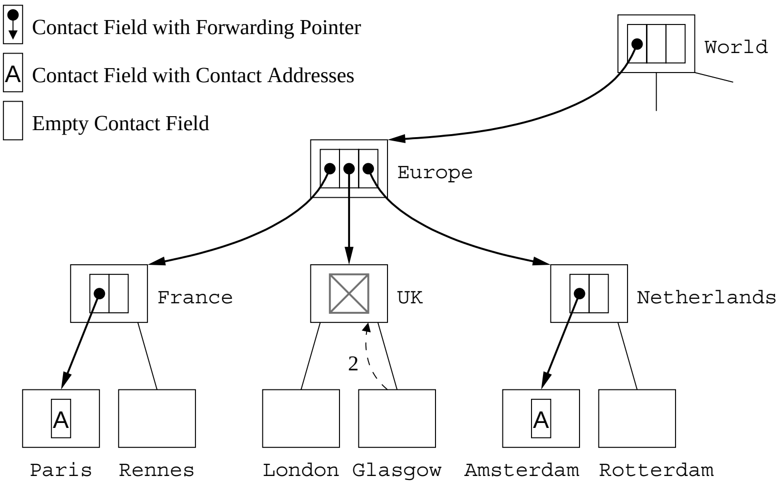


Figure 2.13: In step 2 of the execution of the simplified delete operation, (initiated at the `Glasgow` leaf node), the contact address is deleted at the `UK` node.

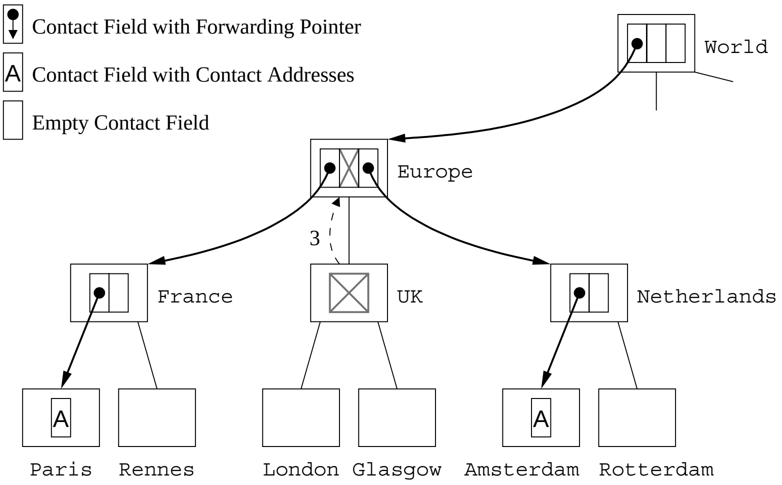


Figure 2.14: In step 3 of the execution of the simplified delete operation (initiated at the Glasgow leaf node), the forwarding pointer is deleted at the Europe node.

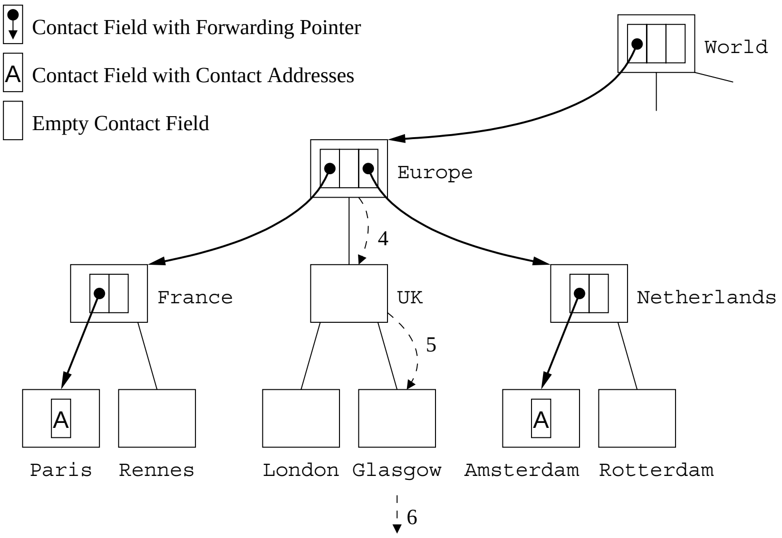


Figure 2.15: In the second half, step 4–6, of the execution of the simplified delete operation (initiated at the Glasgow leaf node), the nodes send their RPC replies.

Like the insert operation, we follow our best-effort approach design principle by performing the delete operation on the contact record database without waiting for the parent's reply. In a real scenario this means that a concurrent downward-going lookup operation might find the path of forwarding pointers pointing to an empty directory node. Since this situation is difficult to avoid without using global locking mechanisms, a more sophisticated lookup operation therefore has to deal with inconsistent search trees.

2.7.3 Implementation

Listing 2.7 shows the procedure *deleteAddress*, used to implement the simplified delete operation. The procedure takes three parameters, as shown in lines 1–3. Object handle *oh* identifies the object of which the contact address set should be modify. Contact address *addr* specifies the address that should be removed. The standard parameter *caller* specifies the directory node that send the *deleteAddress* request. This parameter specifies a child node of the receiving node if the procedure executes at an intermediate node, or the receiving node itself if the procedure executes at a leaf node (i.e., the request came from a client). The *deleteAddress* procedure has no return value. It always succeeds. When the delete operation is ready, the contact address is no longer part of the contact address set. The procedure uses the variable *cr*, defined in line 4, to store and modify the contact record.

The *deleteAddress* procedure starts by retrieving a copy of the contact record from the contact record database, in line 6, using the object handle. If the object handle has a contact record, the contact record is modified in lines 9–23. To modify the contact record, the procedure first checks the contents of the contact field of the calling (child) node. If the contact field stores the contact address, the procedure removes it, in line 11. Otherwise, the procedure checks if the forwarding pointer is set in the contact field, in line 12. If it is set, the procedure clears the forwarding pointer, in line 13.

The changes are committed in lines 16–19. If the contact record has become empty, the contact record is removed from the contact record database, in line 17. Otherwise, the contact record still contains location information, and it is saved back into the contact record database, in line 18.

The procedure then checks, in line 21, whether it has to send a *deleteAddress* request to its parent. If the contact record has become empty and the node has a parent, the forwarding pointer at the parent has become dangling, violating consistency constraint C2. The procedure therefore sends a *deleteAddress* request to the parent, in line 22.

Lines 24-27 deal with the case that the object is unknown at this node. If a parent node exists, the delete operation is continued there (line 26). Otherwise, the delete operation has reached the root node without finding a contact address, and the operation is finished.

```
procedure deleteAddress(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) is
  cr : ContactRecord;
begin
  cr := crDatabase(oh);
  −− Check whether the object is known
  if cr ≠ NIL then
    −− Modify the local contact record
    if addr ∈ cr(caller).addrSet then
      cr(caller).addrSet := cr(caller).addrSet − {addr};
    elsif cr(caller).isPtr then
      cr(caller).isPtr := false;
    end if;
    −− Remove the complete contact record, if needed
    if isEmpty(cr)
      then delete crDatabase(oh);
       else crDatabase(oh) := cr;
    end if;
    −− Tell the parent to remove its forwarding pointer, if needed
    if isEmpty(cr) and parent ≠ NIL then
      call deleteAddress(oh, addr, thisNode) at parent;
    end if;
  elsif parent ≠ NIL then
    −− The object is unknown, try the parent
    call deleteAddress(oh, addr, thisNode) at parent;
  end if;
end deleteAddress;
```

Listing 2.7: The simplified *deleteAddress* procedure.

Chapter 3

Load Distribution

In this chapter we focus on a scalability problem and a related efficiency problem that are most apparent at the root node and its associated domain. While the scalability problem arises from the huge number of contact records located at the root node, the efficiency problem arises from the large geographical area covered by the root domain. We discuss these problems in terms of the root node, but they actually occur in all high-level nodes and domains in the search tree. We present a method that solves both problems by allowing us to distribute the contact records of the overloaded root node over multiple hosts, and to choose the most efficient host to store each contact record.

3.1 Scalability Problems

In the previous chapter we stated that every contact address of every object was reachable from the root node through a path of forwarding pointers. As a consequence, the root node has to store a contact record and handle lookup and update requests for every object currently in use. Given the goal to support 10^{12} objects worldwide, the root node will need to store and handle requests for 10^{12} contact records. This goal puts huge storage and processing requirements on the root node and is obviously a scalability problem.

A simple calculation can quantify these requirements. We first consider the storage requirements of the root node. We assume that the average amount of persistent storage required for an object handle and contact record is 512 bytes. This amount of storage allows us to store the object handle, contact addresses, forwarding pointers, and some administrative information. In this case the total amount of storage space required at the root node is 512 terabytes. Currently, storage space of this magnitude is provided only by off-line storage media, such as magnetic tapes. However, since all the storage space needs to be accessible all the time to deal with lookup and update operations, off-line storage space is clearly not appropriate to store location information.

The ability of the root node to process lookup and update requests fast enough is an even bigger problem. We can look at this from the viewpoint of the insert or the lookup operation. First, we consider the insert operation. If we assume that we want to fill the location service in three years, there are no other update and lookup operations, and most objects have only one contact address, then we need to perform 10^{12} insert operations during those three years. Since every insert operation inserts the first contact address of an object, every insert operation ends up creating a new contact record with a forwarding pointer at the root node. This means that we need to handle 1.1×10^4 insert requests per second at the root. The processing of an insert request should thus take no longer than 95 μs on average.

The lookup operation poses an even bigger problem. Even if every contact record at the root node is accessed on average only once a month by a lookup operation, the root node still needs to handle approximately 3.9×10^6 lookup requests per second. This means that a lookup operation can last at most 0.26 μs! We should note that one lookup access per month per object might be considered a low estimate. Given the storage and processing requirements described above, it is clearly impossible to implement the root node on a single host with current hardware. A scalable implementation of the root node has to consist of multiple hosts.

In the previous chapter we also stated that the root domain covered the whole underlying network. Since the root domain consists of (disjoint) child domains that are distributed all over the world, the child nodes associated with these domains are also distributed all over the world. Using a single location for the root node would therefore imply long-distance communication for at least a subset of its children, resulting in increased use of network resources. This increase is clearly undesirable and forms the basis of our efficiency problem.

Consider, for example, a distributed search tree with the root node located in London and the root node's children representing continents and major countries. In this search tree, lookup and update operations from Australia will need to travel half way around the world to reach the root node in London, incurring communication delays and using network resources. Unfortunately, every other single location for the root node will have the same problem. There will always be child nodes located far away from the single root node. We want to avoid this situation and ensure that the contents of the root node will be available locally to all its child nodes.

3.2 Node Partitioning

The basic solution to both problems is to use multiple hosts to implement the root node. We can solve the scalability problem by distributing the workload of the root node over multiple hosts, and we can solve the efficiency problem by placing parts of the workload at hosts in such a way that it shortens the communication distance. In our discussion we use the term **logical node** to refer to a directory node in the search tree as discussed in the previous chapter. We use the term **physical node** to refer to a host that is part of the implementation of a logical node.

The workload of a logical (tree) node consists of the set of contact records it stores. We partition this set of contact records into disjoint subsets, and assign a subset to each physical node that is part of the implementation of the logical node. Each physical node needs to store and handle the update and lookup requests only for those contact records that have been assigned to it. If we can ensure an even distribution of the contact records over the physical nodes and devise a method to quickly determine at which physical node a contact record is stored, we have solved our scalability problem. We can simply add extra physical nodes when more storage or processing capacity is required.

Figure 3.1 shows the European part of a partitioned search tree with the contact records for one object. In the figure the physical nodes that make up a logical node are depicted inside the logical node. The physical nodes that are responsible for storing contact records for that object (when present) are shown as a rectangle with a solid line; the other physical nodes of a logical node are shown as a rectangle with a dashed line. In the search tree, the logical `Europe` node is implemented using six physical nodes, but only the rightmost physical node of the `Europe` node actually stores a contact record for the object. The logical `France`, `UK`, and `Netherlands` nodes are implemented using two physical nodes each. For the object, the right physical node of the `France` node and the left physical node of the `UK` node store a contact record. If the `Netherlands` node would store a contact record for the object, the right physical node would be responsible for it, as indicated by the solid line. The six leaf nodes all consist of a single physical node. These physical leaf nodes are therefore responsible for storing the contact records of the object, but only the physical node implementing the logical `Paris` leaf node actually stores one.

The independence design principle, as described in Chapter 2, provides the two reasons why contact records form a convenient unit of workload. First, we do not need to maintain *consistency* between contact records of different objects since operations on different objects are unrelated. We can therefore use different physical nodes to handle requests for different contact records, without requiring extra communication to maintain consistency. Second, a contact record is a self-contained data structure; all relevant location information for an object is stored in one place. Contact records can thus easily be moved from one physical node to the next, which is needed when a physical node is added to or removed from a logical node. The process of adding and removing physical nodes is described in Chapter 7.

Since communication with a logical node, such as a parent or child, actually means communicating with a specific physical node of the logical node, we need a way to determine at which physical node the contact record of an object is (or should be) stored. This determination should be done quickly, preferably without communicating with other hosts, to avoid slowing down the actual communication. The object handle contains specific information to aid in this determination. The object handle thus consists of two parts: an object identifier and a selection field. This is shown in Figure 3.2. The **object identifier** is solely responsible for identifying the object, but is an invisible part of the object handle in most of the location service. The extra information needed to quickly determine which physical node to use is stored in the **selection field** of the object handle. The exact use and internal structure of the selection field is explained later in this chapter.

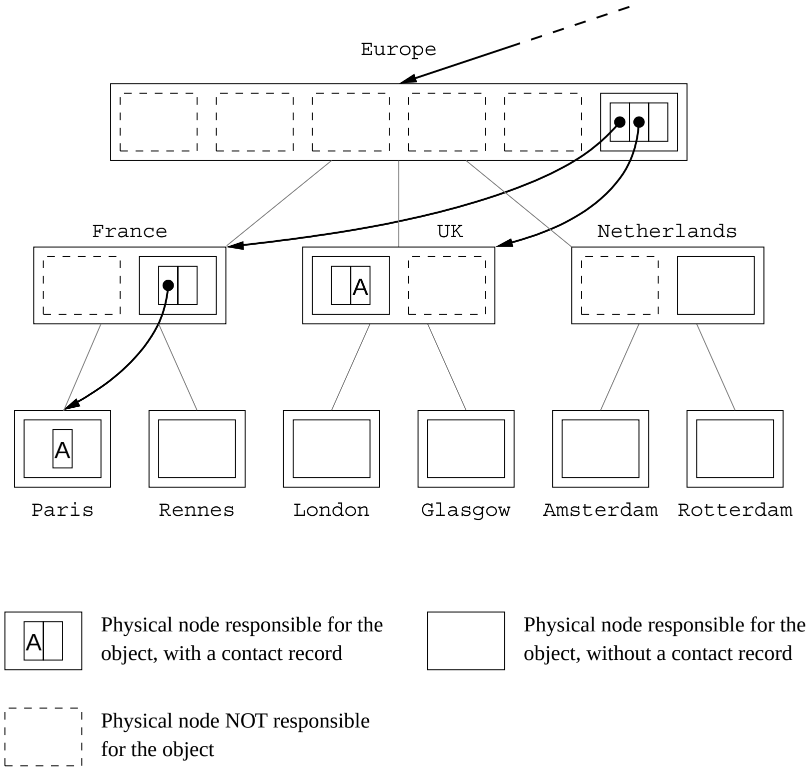

Europe
France
UK
Netherlands
A
A
Paris
Rennes
London
Glasgow
Amsterdam
Rotterdam
A
Physical node responsible for the object, with a contact record
Physical node responsible for the object, without a contact record
Physical node NOT responsible for the object

Figure 3.1: The European part of the search tree with logical tree nodes partitioned into physical nodes. Only four of the physical nodes that are responsible for storing a contact record for the object actually store a contact record.

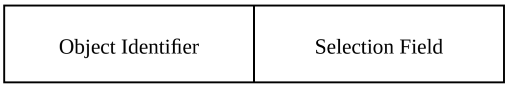


Figure 3.2: Two parts of an object handle.

We can estimate the number of physical nodes needed to implement the (logical) root node in the three cases described above using the hardware characteristics of current hardware. We first look at the storage requirements. If a physical node is capable of efficiently storing up to 500 gigabytes, we require at least 1,000 physical nodes to implement the root. In the second case we look at the insert requirement, that is, filling the location service in three years. If we assume that handling an insert request consists mostly of reading a random contact record from disk and writing it back synchronously, an insert operation will last on average 30 milliseconds (assuming typical disk properties in 2002). In this case we require at least 300 physical nodes to implement the logical root node. In the third case the root receives one lookup request per contact record per month. If we assume that a lookup request mainly consists of reading a random contact record from disk, a physical node can handle a lookup request in on average 15 milliseconds. In this case we require around 58,000 physical hosts to implement the root. Given these calculations, we expect the root node may grow till 10^3 to 10^5 physical nodes.

3.3 Nearby Communication

In the rest of the chapter we deal with the problem of choosing a physical node to store a contact record of a particular object in such a way that it avoids long-distance communication for that object. By devising such a method, we solve the efficiency problem that deals with long-distance communication at the root of the search tree. The main requirement for choosing a placement strategy is that we desire communication between *nearby* physical nodes where possible. However, this requirement should not interfere with the main scalability requirement of evenly distributing the load of the logical node over its physical nodes. To simplify our discussion, we assume all physical nodes have the same storage and processing capabilities. Before we can discuss the placement of contact records at physical nodes, we first need to discuss the placement of physical nodes in the network.

3.3.1 Physical Node Placement

We have, in principle, complete freedom in where in the network to place the physical nodes that make up a logical node. However, the most logical place for these nodes is within the domain of the logical node they implement. For example, the physical nodes that implement the `Europe` node are located within the European domain. That way the European physical nodes are at least near the physical nodes that implement the child nodes, for example, the physical nodes implementing the logical `France` node. If the domain associated with a logical node is small, there is not much choice in where to place its physical nodes. If we consider the root domain, however, we can place its physical nodes all over the world.

We use a two-phase strategy to place physical nodes in a domain. In the first phase, we distribute physical nodes evenly over the domain. This phase ensures that there is always a physical node in the proximity of a child node. There are, however, places in the domain where more objects are located than in other places, for instance, in and around densely populated areas, such as cities. In the second phase, we therefore add extra physical nodes in those places in the domain to increase the local storage and processing capacity. The second phase is necessary to avoid overloading local physical nodes. In short, we ensure that there is a minimum density of physical nodes in the domain, with increased density where needed. To determine the right number and placement of physical nodes for a logical node, the location service needs to gather statistics on the origin and frequency of lookup and update operations (see also Chapter 7).

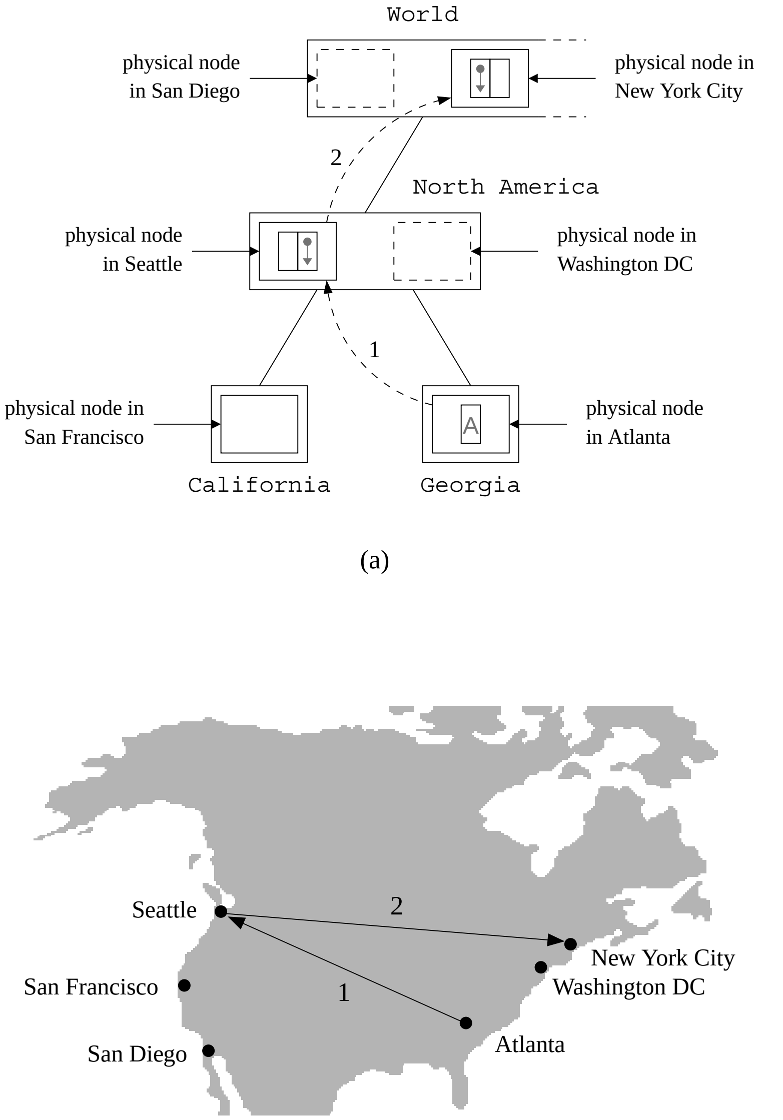
3.3.2 Hashing

A naive (but effective) approach to placing contact records at physical nodes is to assign them randomly to physical nodes. This can be done by inserting a random value in the selection field of the object handle, and using this value to index the list of physical nodes. This is basically a hashing scheme. This approach has excellent load-balancing characteristics since it can provide a uniform work distribution. It therefore provides a basic solution for our scalability problem.

Unfortunately, this approach has poor communication patterns in large domains. Consider a three level search tree with a state (or country), continent, and world level, and focus on the part of the search tree that covers North America. This search tree is shown in Figure 3.3(a) and the geographical placement of its physical nodes is shown in Figure 3.3(b). In the search tree, there are two leaf nodes, located in Atlanta and San Francisco that both consist of a single physical node. Their parent, the logical `North America` node, consists of two physical nodes, one located in Seattle and one in Washington DC. The logical root node consists of a number of physical nodes distributed worldwide with one of them located in New York City and another located in San Diego. Now consider what could happen when the first contact address of a new object is inserted at the `Atlanta` leaf node and its contact records would be placed at physical nodes using the hashing scheme described above.

To create a consistent search tree, forwarding pointers need to be inserted at the logical `North America` and `World` nodes. Let us assume that for the logical `North America` node, the selection field of the object handle hashes to the physical node in Seattle resulting in a contact record with forwarding pointer being stored at the `Seattle` node. For the logical `World` node, the selection field hashes to the physical node in New York City, and the root contact record is stored there. With this scheme, the insert operation therefore visits physical nodes in Atlanta, Seattle, and New York City (in that order) to insert the contact address and create a path of forwarding pointers.

This example shows an inefficient communication pattern, performing first an RPC from the east coast to the west coast (step 1 in the figure) and then back again (step 2 in the figure). We would like to avoid this erratic crisscross pattern. In fact, we simply want to use the physical node in Washington DC, not the physical node in Seattle. In more general terms, for a contact address that is inserted in Atlanta, we would like to use physical nodes in the general vicinity of Atlanta. To ensure this, we need a load-distribution scheme



(a)

(b)

Figure 3.3: An insert operation crosses North America twice while it is going up in the search tree. Note that the forwarding pointers in figure (a) are drawn small to simplify the figure.

that promotes short-distance communication by placing contact records at neighboring physical nodes, instead of placing the contact records at random.

3.3.3 Forcing Locality

To explain our method for ensuring nearby communication, we look at communication patterns in the search tree, specifically at the geographical locations of the physical nodes involved in an operation and the order in which they are used. We first look at the communication pattern caused by the insert operation of the first contact address of an object, and show how our method shortens the communication distances. We then generalize our solution to include all possible communication patterns caused by the insert, delete, and lookup operations.

First Insert Communication Pattern

When inserting the first contact address of an object, the insert operation creates new contact records at every logical node on the path from the leaf node up to and including the root node. The central notion of our placement method is that we want these newly created contact records stored at physical nodes that are *geographically* near to each other. By using physical nodes that are geographically close by, we can avoid using long-distance networks and keep the distance traveled during the insert operation small. This approach improves the efficiency of our location service by keeping the use of network resources low.

We make the assumption that a large **geographical distance** between physical nodes implies a large **network distance**. By avoiding communication over large geographical distances, we reduce the risk of communicating over large network distances. There is, of course, still the possibility that two geographically close physical nodes will have a large network distance due to some inefficient local network topology. We ignore such pathological cases. We assume that in current wide-area networks our assumption is realistic when talking about large distances, for instance, in the order of a 1,000 km or more.

We expect that this assumption will become valid for smaller distances in the future for the following three reasons. First, computer networks are becoming increasingly prevalent. Second, existing networks are becoming increasingly inter-connected, as has happened in the public phone system in the past. Third, with the increased use of fiber optics for nonlocal communication and the decreasing delays in the nonfiber part (e.g., signal amplifiers) of these communication networks, the propagation speed in computer networks is approaching the speed of light. The network latency will thus be increasingly dominated by the actual geographical distance traveled. We revisit this assumption in Section 3.6.

The reason for trying to shorten the communication distance using geographical distance as a distance metric, instead of using some form of network distance directly, is based on the notion that the information used to compute distances should be *durable*. Since objects are allowed to exist for long periods of time, we can expect object handles

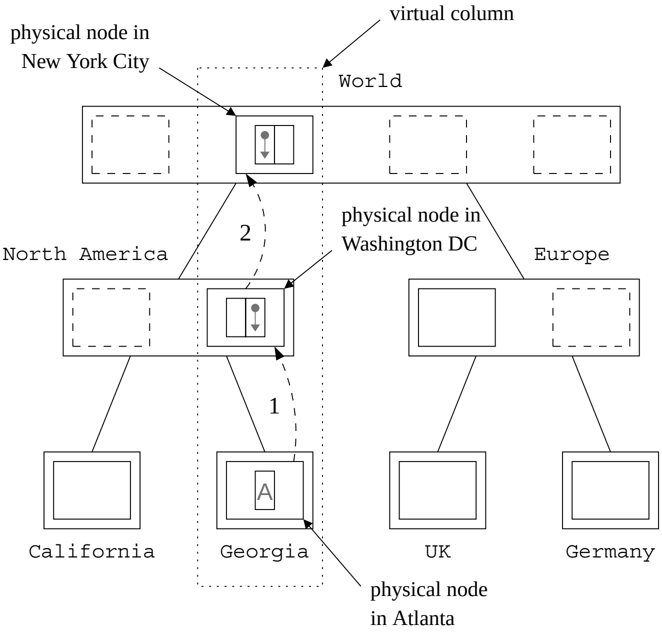


Figure 3.4: A virtual column in a partitioned search tree formed by the physical nodes used to store the contact records for one object located in Georgia.

and the information in their selection fields to have a life span longer than a single configuration, and maybe even implementation, of the location service. The information used to compute distances should therefore be usable in different configurations of the search tree and the underlying network in general. Geographical locations and distances provide a stable distance metric since the distances between geographical locations do not change over time, at least not significantly.

By placing the new contact records, located at different levels in the tree, at physical nodes that are in each other's general vicinity, we create a kind of **virtual column** through the tree, as shown in Figure 3.4. This virtual column is formed by the leaf node where the contact address was stored combined with the intermediate and root physical nodes where the forwarding pointers are stored. In our example, the location of the virtual column is determined by the `Atlanta` leaf node, and the virtual column is thus located on the east coast of North America.

We create the virtual column by having the geographical location of the leaf node determine the physical nodes used at every logical node on the path from leaf to root. We do this by placing the geographical location of the leaf node (i.e., its longitude and latitude) in the selection field of the object handle. When a client creates a new object handle, it takes the geographical location of the leaf node where the first insert will take place and provides that location to the procedure that creates the new object handle.

The placement strategy then places a contact record at the physical node closest to this location. When a physical node wants to send an RPC request to a logical node, it uses the placement strategy to determine which physical node is responsible for storing the requested contact record and to which physical node it should thus send its request message. The placement strategy is implemented by computing the geographical distances between the geographical location in the selection field and every physical node of a logical node and selecting the physical node with the shortest distance. An efficient way to determine the physical node closest to the location in the selection field is described in Section 3.5

General Communication Patterns

The virtual column notion is specific to the insert operation of the first contact address of an object since it uses the geographical location of the initial leaf node. The notion of avoiding the crisscross communication pattern by using the nearest physical node is more general, however, and can be applied to all communication in the search tree. The vicinity requirement can be generalized informally by saying that the communication between levels in the tree should not switch *geographic direction* when communicating over longer distances.

If a leaf node and physical root node are far apart, the geographical path traveled from the leaf to the root node should always go toward the geographical location of the physical root node. In other words, an operation should *home in* on the physical root node. This is the most direct (i.e., shortest) route between the leaf node and physical root node. By going up one level in the search tree, the physical node used at the higher level should either be closer to the object's physical root node, or stay in the general vicinity of the physical node at the lower level. This means that we would like the physical nodes used to be on or near the line connecting the locations of the physical leaf node and physical root node.

Consider, for example, the search tree shown in Figure 3.5. The object handle used in this figure stores in its selection field the geographical location of Atlanta. The physical node in New York City is therefore selected to store a contact record for the logical `World` node. In the figure a client near the `Germany` leaf node wants to insert a contact address. The contact address is stored at the physical node in Berlin since the leaf node consists of only a single physical node. For the logical `Europe` node, our placement strategy must choose between the physical nodes in London or in Moscow. It selects the physical node in London since that physical node is closest to Atlanta. The physical node in London communicates, in turn, with the physical root node in New York City. The insert operation thus travels toward the west.

It is important to realize that by adding a geographical location to the object handle we did not endanger the object handle's location independence nor the inherent locality of the search tree. The object handle can still be used to insert contact addresses at every leaf node in the search tree. The location is used solely to determine the most efficient place in the search tree to store the data. The inherent locality of the search tree is not

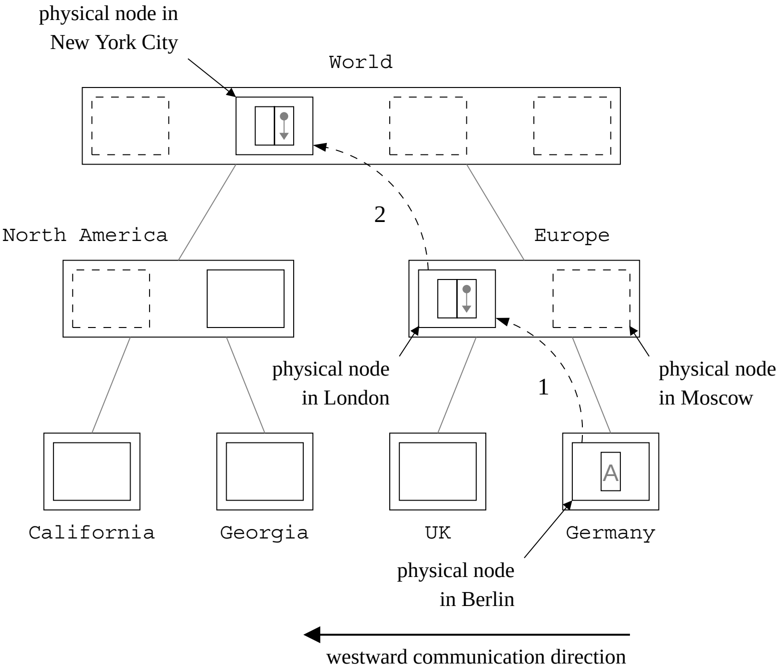


Figure 3.5: The communication direction toward the west of an insert operation while it is going up in the search three. Note that the leaf node and physical root node are located far away from each other.

endangered since it is determined by the hierarchy of domains, not where in the domain a physical node resides. From the viewpoint of the (logical) search tree, the geographical location is just some random bits of the object handle.

The choice of which geographical location to place in the selection field was obvious for the virtual column case. For the general case, however, it is less clear which location to use. The geographical location is in fact a *hint* on where the object generally resides in the world. The efficiency of our placement scheme depends on the accuracy of this hint. In our current system, we use the geographic location of the leaf node where the first insert took place. We consider the use of the location of this leaf node, however, only a heuristic that a client should be able to overrule when it knows that another location is more appropriate.

3.3.4 Internal Structure of the Object Handle

As mentioned before, the object handle consists of two parts: the object identifier and the selection field. The internal structure of the object handle is shown in Figure 3.6. The

Object Identifier	Selection Field		
	Longitude	Latitude	Random

Figure 3.6: Internal structure of an object handle.

object identifier is (still) responsible only for identifying the object, and can be considered an opaque bitstring. The selection field consists of three fields: longitude, latitude, and random. The longitude and latitude specify the geographical location of the object, that is, the location where the first contact address was inserted. The random field contains the value used to break ties when multiple physical nodes use the same location, which is explained next.

When we discussed the placement of physical nodes in the network in Section 3.3.1, we discussed the possibility that there could be too much load at a single location and that we could add extra physical nodes to deal with this increased load. However, since these extra physical nodes would share the exact same location with the original physical node, our placement strategy is not able to distribute the load among them. We therefore keep the random value in the selection field. Using the hashing scheme, the random value allows us to distribute the load evenly over physical nodes that share the same location. We adapt our location-aware load-distribution scheme as follows. It first takes the geographical distances into account to select the physical node nearest to the location in the selection field. When multiple physical nodes share the same location, the strategy uses the random value to select a single physical node.

When the selected geographical location is placed in the selection field, the object handle is fixed and cannot be changed anymore. This inability to change the object handle might become a problem when many objects change the location where they reside permanently since their virtual column will no longer be in the right place. We think we can solve this problem by associating a lease [Gray and Cheriton, 1989; Duvvuri et al., 2000] with the location information in the object handle to force clients to keep the object handle up-to-date, but we have not looked at this solution in detail.

3.4 Simulation Results

To investigate the load-balancing and distance-shortening characteristics of the location-aware load-distribution scheme, we performed a simulation experiment. In the experiment we measured the workload experienced by physical nodes and the geographical distance traveled by lookup and update operations. In the investigation, we also measured the load and distances resulting from other load-distribution schemes, and compared these results with the load and distances resulting from the location-aware load-distribution scheme. To ensure a fair comparison between the various load-distribution schemes, we used the same logical search tree and the same number of physical nodes to implement logical nodes in all simulations. The hypothesis of the experiment was that the location-aware load-distribution scheme leads to short communication distances and a reasonable load balance. To investigate the general usefulness of the hierarchical location service approach, we also compared the best performing load-distribution scheme with the home-based location service approach.

3.4.1 Methodology

To perform our simulation experiment, we had to decide on several aspects of the experiment, such as, what tree structure to use and what quantities to measure. In this section we look at these aspects.

Tree Structure

We constructed a real-life search tree using the World Cities Population Database[1] (WCPD) [Rhind, 1991]. This database lists all the country capitals and those cities in the world with more than 100,000 inhabitants in the year 1987. For every city the database stores its population, country, and geographical coordinates. In our experiment we used only cities with more than 100,000 inhabitants (capital or not), resulting in 2,299 cities with a total population of little under a billion.

Using the WCPD, we constructed a (logical) search tree consisting of four levels: a single root domain, eight subcontinent domains, 122 country domains, and 2,299 city domains. Since no subcontinent information was given in the database, we grouped countries into subcontinents by hand. Table 3.1 shows the characteristics of the subcontinent domains.

Every city in the database had a corresponding leaf domain and also contributed a physical node to every domain it was a part of. For example, the city of Amsterdam had a leaf node and contributed a physical node to the logical `Amsterdam`, `Netherlands`, `Europe`, and root nodes. Since there are 2,299 cities and every city supports four physical nodes (i.e., one for every level in the tree), the search tree consists of 9,196 physical nodes in total. In the simulation, we assumed that all physical nodes had the same storage and processing capabilities.

Load-distribution Schemes

In the experiment we compared four load-distribution schemes: the hashing scheme, the location-aware scheme, a hybrid hashing/location-aware scheme, and the logical scheme, as shown in Table 3.2. The hashing scheme used the random value in the selection field of the object handle to pick a physical node from a logical node. The location-aware scheme used the geographical location in the selection field of the object handle to pick the nearest physical node in a logical node. To see if we could get the best characteristics of both these schemes (i.e., load distribution and distance shortening), we also added a hybrid hashing and location-aware scheme. In this hybrid scheme, we used the location-aware scheme at

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

Table 3.1: Information on the subcontinent domains used in the simulation experiment.

Subcontinent	Countries	Cities	Population
North America	11	274	98,029,259
South America	13	240	96,418,651
Europe	29	436	142,158,983
Africa	32	124	40,157,801
East Asia	4	493	271,493,831
South Asia	17	336	130,192,023
North Asia	2	287	112,019,100
Australasia	14	109	49,271,682
Total	122	2,299	939,741,330

Table 3.2: The four load-distribution schemes examined.

Mapping	Description
Hashing	For domain *D*, select a city in *D* using the random value in the object handle.
Location-aware	For domain *D*, select the city in *D* closest to the geographical location in the object handle.
Hybrid	Use the location-aware mapping for level 0 and 1 and the hashing mapping for level 2 and 3.
Logical	For domain *D*, select the city nearest to the center of domain *D*.

the root and subcontinent levels and the hashing scheme at the country and city levels. We added the logical scheme to measure the load and distances of a nondistributed scheme. In the logical mapping scheme, we used the city nearest to the center of the associated domain to store all contact records of the logical node. We determined this center by computing the distances between all the cities in a domain and choosing the city with the smallest aggregated distance to all other cities.

Workload Generation

In the simulation, activity was generated by two types of events: an object moving and a client looking up a contact address. Since we assumed these events to be unrelated, we used two separate patterns to model objects: a mobility and a lookup pattern. The mobility and lookup patterns chosen for a specific object determined at which leaf nodes its update or lookup operations were initiated. We used two types of mobility and lookup patterns:

a global and a local pattern. Objects with a global mobility and lookup pattern move over long distances and have clients that are located far away; objects with a local mobility and lookup pattern move only over small distances and have clients that are located nearby.

We created the global and local pattern using the following method. We started by associating a mobility probability and a lookup probability with every level in the search tree. To determine the destination of a move operation given a starting location, we used the mobility probabilities to randomly pick a level in the tree and determined the domain that contained the starting location at that level. We then randomly chose a city in that domain as the destination for the move, using a uniform distribution. For example, if our object resided in Amsterdam and we chose for the move the subcontinent level at random, the destination address would be a city in Europe, for instance, Madrid.

To determine the location of a client performing a lookup operation given the location of the object, we used a similar method. We used the lookup probabilities to randomly pick a level and determined the domain in which the object resided at that level. We then randomly chose a city in that domain as the client location, using a uniform distribution weighted by the population size of each city. For example, if our object resided in Madrid and we chose for the client location the root level at random, the location of the client would be a city in the world. However, the big cities, such as Mexico City and Tokyo, would be more likely to be chosen than small cities.

To simplify our simulation, we chose three combinations of mobility and lookup patterns, and called these combinations object models. Table 3.3 shows the object models with their associated probability distributions. The combination GM-GL (Global Mobility - Global Lookups) models a globally moving object with globally distributed clients. The combination LM-GL (Local Mobility - Global Lookups) models a locally moving object with globally distributed clients. The combination LM-LL (Local Mobility - Local Lookups) models a locally moving object with locally distributed clients. In the table, level 0 refers to the root level and level 3 refers to the leaf level. The global distribution is created by associating the same probability with all four levels (i.e., 25%), making all levels equally likely. The local pattern is created by associating larger probabilities with the lower level (e.g., 50% for the leaf level) and smaller probabilities with the higher levels (e.g., 10% for the root level), making the smaller domains more likely to be picked.

In the experiment we focused on the load and distances generated by a simple mobile object. Since we did not consider replicated objects, all objects had only a single contact address. Furthermore, to best measure and compare the resulting load and distances, we did not use any optimizations, for instance, such as those described in Chapter 4. Using optimizations would make the comparison of simulation results more difficult. Contact addresses were therefore always stored in leaf nodes and no caching was done during lookup operations.

Quantities Measured

In the simulation we looked at the load experienced by individual physical nodes, the total geographical distance traveled by update and lookup operations, and at the frequency

Table 3.3: The probability distributions of the three object models used in the simulation experiment. G refers to global domains, and L refers to local domains. Model LM-GL thus stands for mobility only in the local domain and lookup operations initiated in the global domain.

Model ↓	Mobility				Lookup			
Level →	3	2	1	0	3	2	1	0
GM-GL	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
LM-GL	0.50	0.25	0.15	0.10	0.25	0.25	0.25	0.25
LM-LL	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10

distribution of the distances traveled by operations. The simulation ignored the dynamic behavior of the network (e.g., congestion) since we were interested only in the overall load distribution and the distance traveled.

We measured the load in a physical node by the number of update and lookup procedures executed. To simplify the simulation, we combined the load of the four physical nodes in a single city and recorded only the combined load per city. We measured the geographical distance traveled (in kilometers) as the combined distances between physical nodes involved in the operations. To measure the frequency distribution, we maintained a frequency table indexed by distance. The size of the bins in this table was 100 km, that is, the first bin recorded the frequency of distances between 0 km and 100 km.

Execution of the Simulation

We performed the simulation as follows. For every object model, we generated 100 million events, with an event either being a move operation or a lookup operation. For every event, we first determined whether it was a move or lookup operation. In the simulation we used a fixed ratio of one move operation per four lookup operations. We then randomly picked the home location of the object used by the operation. The city used as home location was picked using a uniform probability distribution weighted by the populations of the cities.

If the event was a move operation, the source and destination cities of the move needed to be determined. The source city was determined by using the mobility pattern and the home location of the object as the starting location. The destination city was also determined by the mobility pattern, but in this case using the source city as the starting location. When source and destination of the move were known, we simulated an insert operation starting at the destination city and a delete operation starting at the source city. We determined the physical nodes used by the update operations, and registered the execution of an update procedure at these nodes. We also determined the geographical distances between these physical nodes, and added these distances to the total geographical distance covered by update operations. We also used the geographical distances to update the frequency distribution table.

If the event was a lookup operation, the locations of the object and the client needed to be determined. The location of the object was determined by using the mobility pattern and the home location of the object as the starting point. The location of the client was determined by the lookup pattern and using the location of the object as the starting point. When locations of the object and the client were known, we simulated the lookup operation. As for a move operation, we determined the physical nodes used by the lookup operation, and registered the execution of a lookup procedure with those nodes. We determined the geographical distances between the physical nodes, and added those to the total geographical distance covered by lookup operations. We also used the geographical distances to update the frequency distribution table.

3.4.2 Results

We compare the results of the various load-distribution schemes by first considering their load-balancing characteristics. We then examine their distance-shortening characteristics.

Load Balancing

To examine the load-balancing aspect, our simulation program generated a table listing the total number of executed procedures per city for each load-distribution scheme. To best compare the different load-distribution schemes, we sorted this table starting with the heaviest loaded city and plotted the percentage of the total load experienced by the first *N* cities in the table. The resulting graphs are shown in Figures 3.7–3.9. Since the figures show similar results, we focus on the results depicted in Figure 3.7.

A perfect load balancing scheme where all physical nodes are evenly loaded would result in a straight diagonal line from 0% load at 0 cities to 100% load at 2,299 cities since every city would add the same amount of load. On the other hand, a load-distribution scheme that results in vast differences in the load experienced by different cities, would result in a strongly arched graph since heavily loaded cities would be sorted to the start of the table. To ease the understanding of the figures, we also indicate the number of cities that support 50% of the load. A perfect load balancing scheme would have 1,149 cities experiencing 50% of the total load.

In all three figures the hashing scheme results in the arc that is nearest to the diagonal. This scheme thus has the best load-balancing characteristics, as expected. Since only 753 cities experience 50% of the total load (instead of 1,149), it still does not provide perfect load balancing. The reason for this imbalance is the fact that the client locations for lookup operations were chosen using a uniform distribution weighted by population size. Leaf nodes of big cities started therefore more lookup operations than leaf nodes of small cities, leading to the imbalance. As expected, the logical load distribution (i.e., no load distribution) performs the worst, with only 19 cities supporting 50% of the load.

The location-aware load-distribution scheme does not have a particularly good load distribution with only 245 cities supporting 50% of the load. It is, however, still better than the logical distribution scheme. As conjectured, the hybrid location-aware/hashing

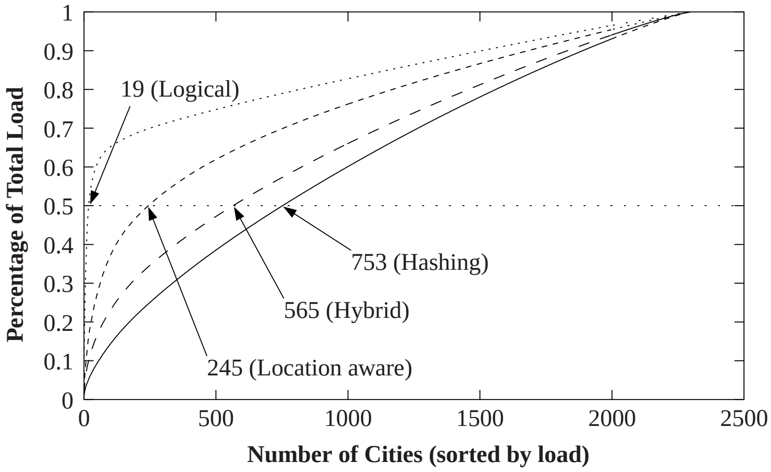


Figure 3.7: Load distribution generated by globally distributed lookup operations on a globally moving object (Model GM-GL). The graph shows the total load experienced by the first *N* cities.

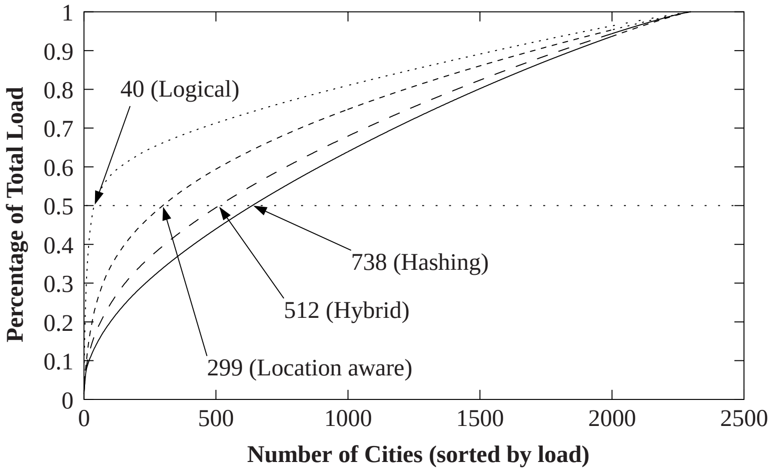


Figure 3.8: Load distribution generated by globally distributed lookup operations on a locally moving object (Model LM-GL). The graph shows the total load experienced by the first *N* cities.

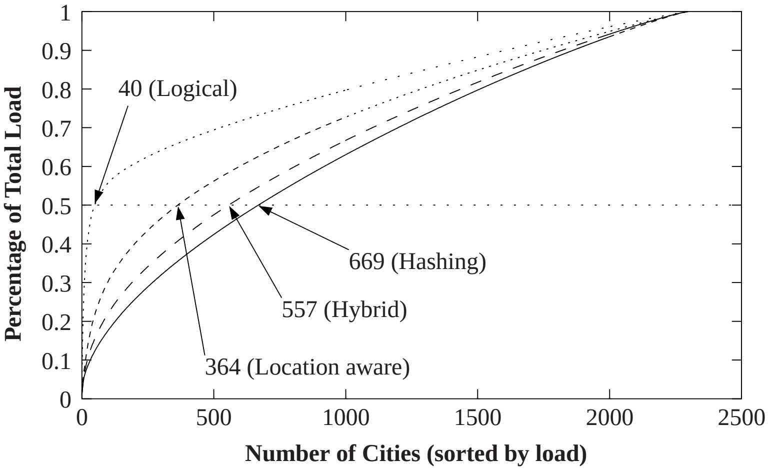


Figure 3.9: Load distribution generated by locally distributed lookup operations on a locally moving object (Model LM-LL). The graph shows the total load experienced by the first *N* cities.

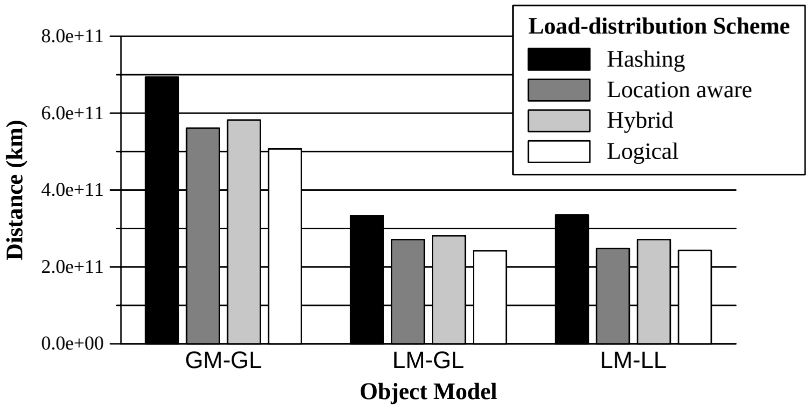
scheme has a load-balancing effect that falls between the location-aware and hashing schemes, with 50% of the total load experienced by 565 cities.

Distances Traveled

To examine the distance-shortening characteristics and the resulting locality, our simulation program also recorded the total geographical distance traveled by all operations and the number of operations that traveled a particular distance. Figure 3.10 shows the resulting total geographical distance traveled. The percentages in the table of Figure 3.10 are relative to the distances of the hashing scheme.

As expected, the hashing scheme results in the largest total distance traveled for all object models. The location-aware scheme improves this distance with 19% and 26% depending on the object model. The improvement of the location-aware scheme is stronger for the local object model since all operations are initiated around the home location, as explained in the previous section. Since most operations are handled locally in smaller domains, the effect of the improvement at the root level is not diminished by other operations initiated far away, as is the case with, for instance, the LM-GL object model.

It is less obvious why the logical scheme results in the shortest total distance, that is, shorter than the location-aware scheme. The main reason is that the logical scheme makes a conservative choice when choosing the city to place a contact record. By placing contact records at the city in the center of a domain, it avoids choosing a city located far away from other cities at the center of their domains, such as the parent or child domains. Note also that the location-aware scheme is most useful with stationary objects since it



Model	GM-GL		LM-GL		LM-LL	
	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
Hashing	6.9e11	100%	3.3e11	100%	3.4e11	100%
Location aware	5.6e11	81%	2.7e11	81%	2.5e11	74%
Hybrid	5.8e11	84%	2.8e11	84%	2.7e11	81%
Logical	5.1e11	73%	2.4e11	73%	2.4e11	73%

Figure 3.10: The total absolute and relative geographical distances traveled by update and lookup operations using the three object models and different load-distribution schemes.

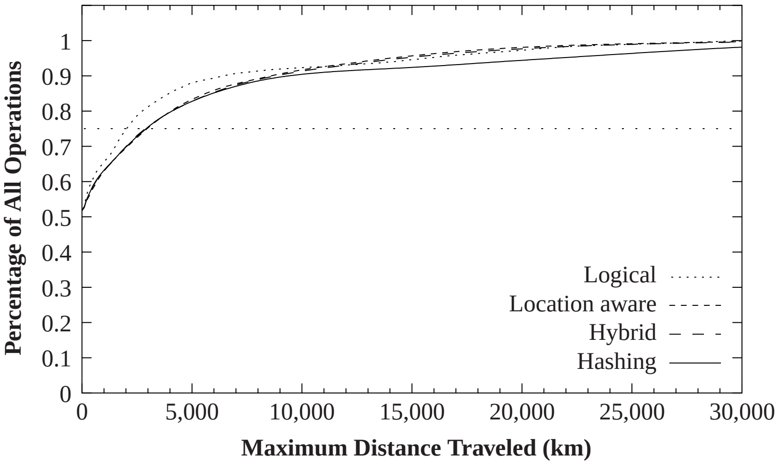


Figure 3.11: Distribution of the distances traveled by lookup operations showing differences in locality (Model LM-GL).

most strongly optimizes the first insert operation. Since the simulation does not consider completely stationary objects, that improvement is not visible.

The hybrid scheme results in distances comparable to the location-aware load-distribution scheme. It can therefore indeed be considered as an attractive compromise between load balancing and distance shortening. Unfortunately, the optimization of the total distances with the hybrid scheme is relatively small: between 16% and 19% for our object models.

To examine the locality of the search tree with the different schemes further, we also plotted the distribution of the distances traveled by operations. Figure 3.11 shows the distance distribution of lookup operations for the LM-GL object model. We show only a figure for this model since the figures for the other models are similar. The figure plots the accumulative number of operations for a given distance, as a percentage of the total number of operations. Strong locality is visible as a high percentage of the operations at small distances since that means that a large number of operations needed to travel only a short distance. Weak locality would result in a slowly rising percentage since only few operations would travel a short distance. To ease the interpretation of the figure, we also added the 75% mark.

The distribution of distances is roughly the same for all schemes, with the logical scheme performing only slightly better. Apparently, for the types of objects simulated, the locality of the search tree is not very dependent on the load-distribution scheme. With the logical scheme, 75% of the lookup operations traveled less than 2,200 km; for the other schemes, the operations traveled less than 3,100 km. Only 10% of the lookup operations traveled more than 6,500 km. Since the circumference of the Earth at the equator is slightly over 40,000 km, the operations experience reasonable locality.

Conclusion

From the simulation results it is clear that the location-aware load-distribution scheme leads to shorter communication distances. It does, however, not lead to a reasonable load balance. If we also want to ensure a reasonable load balance, we need to use the hybrid load-balancing scheme.

3.4.3 Comparison with the Home-based Approach

To examine the effectiveness of our hierarchical location service in general, we also compared our location service using the hybrid scheme with the results of a home-based location service. In a home-based location service every object has a single server at the object's home location that stores its current location (i.e., a contact address). Home-based location services are used in Mobile IP [Perkins, 1998] and mobile phone systems [Mohan and Jain, 1994]. In our simulation every city provided a home location for its objects. This home location needed to be contacted for all lookup and move operations.

Load Distribution

We first compare the differences in load balancing. Figure 3.12 shows the load balancing for both types of location services. As in the previous load-balancing figure, we added the 50% mark. The figure shows that the hierarchical location service has better load-balancing characteristics. This figure is, however, somewhat misleading. The reason is that the absolute load is smaller in the home-based location service. This is clearly visible when comparing the absolute load, as shown in Figure 3.13. This figure shows that even though the load balance might be less, the load per city is smaller for the home-based. The reason that the hierarchical location service generates more load is that it consists of four times as many nodes as the home-based location service.

Distances Traveled

Figure 3.14 shows the total distance traveled by update and lookup operations for both types of location service. Depending upon the object model, the hierarchical or the home-based location service performs better. The home-based approach performs better when there is no locality in the request or very high locality (object model GM-GL and LM-LL, respectively). This is expected, with no locality all operations are global, and no locality can be exploited by the hierarchical location service; with very strong locality, we can place the home location in the area where the requests come from, allowing the home-based location service to exploit locality. The hierarchical location service provides better performance when there is variable locality, as with object model LM-GL.

Figure 3.15 show a comparison of the locality of lookup operations of the LM-GL object model for the two types of location services. An important difference between both types is that the home-based location service has a maximum distance that can be traveled of around 20,000 km, which is half the circumference of the Earth. This is apparent in the

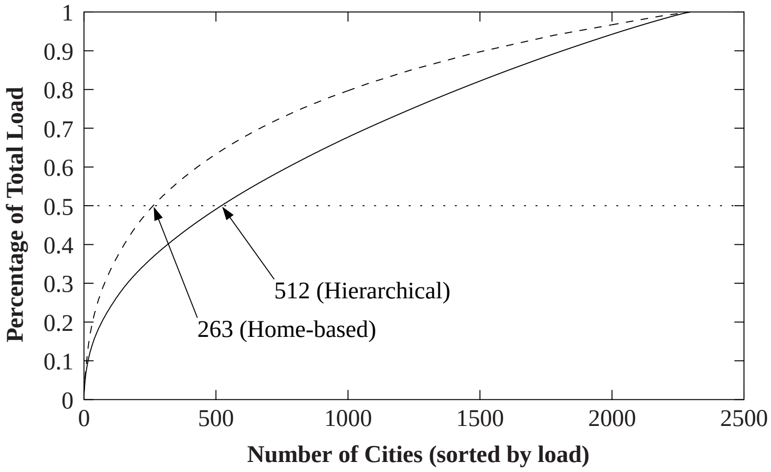


Figure 3.12: Comparison of the load balancing of the hierarchical and home-based location services showing differences relative to the total load (Model LM-GL).

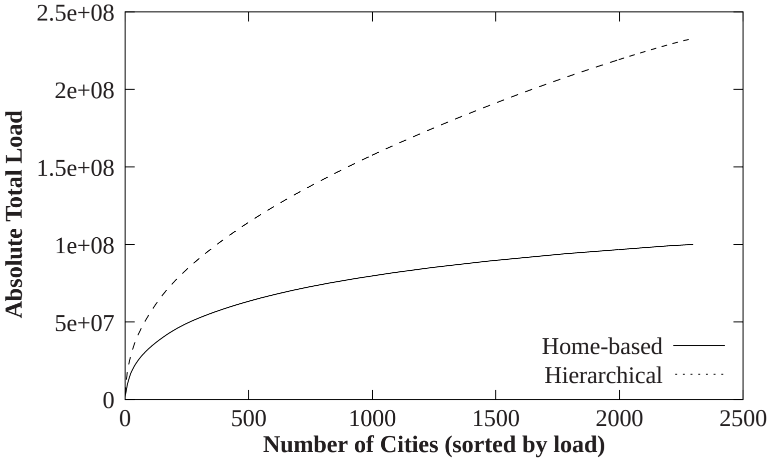
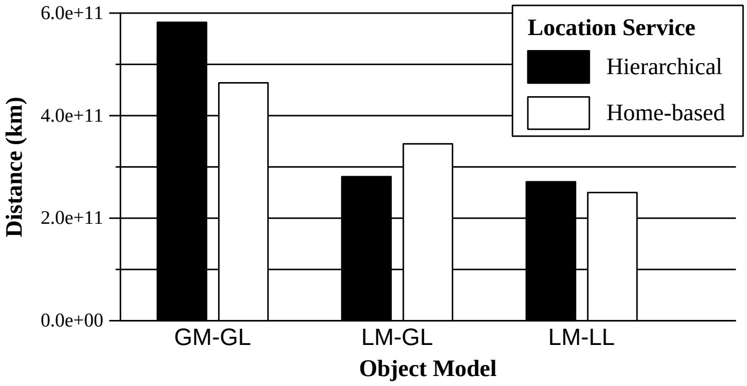


Figure 3.13: Comparison of the load balancing of the hierarchical and home-based location services showing absolute differences (Model LM-GL).



Model	GM-GL		LM-GL		LM-LL	
	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
Hierarchical	5.8e11	100%	2.8e11	100%	2.7e11	100%
Home-based	4.6e11	80%	3.5e11	123%	2.5e11	92%

Figure 3.14: Comparison of the total distance traveled by operations in the hierarchical and home-based location services.

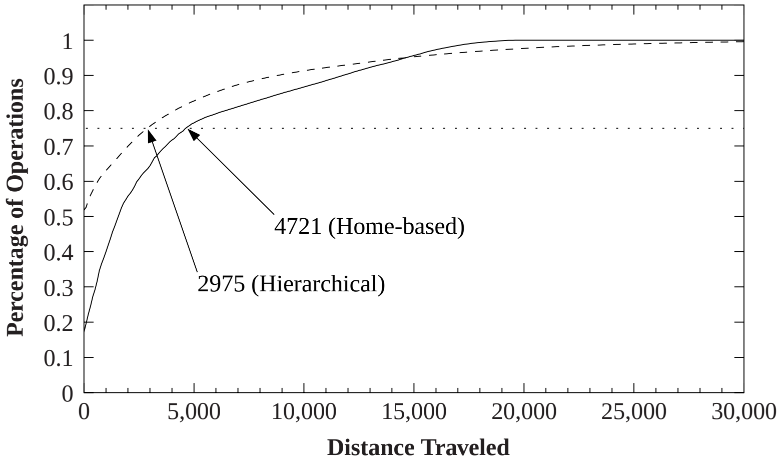


Figure 3.15: Comparison of the locality of the hierarchical and home-based location services using the distribution of the distances traveled by lookup operations (Model LM-GL).

figure by the fact that for the home-based approach all (100%) distances are 20,000 km or less. In contrast, the hierarchical location service has a few distances larger than 20,000 km. The hierarchical location service has better locality, that is, more lookup operations travel a shorter distance. For instance, 75% of the lookup operations travel up to 2,975 km in our hierarchical location service but up to 4,721 km in the home-based location service. This is not as bad as it seems since only 13% of the lookup operations in the home-based approach travel between 2,975 km and 4,721 km.

Conclusion

From the simulation results it is unclear whether our hierarchical location service performs better. Given the object models simulated, the hierarchical location service seems to perform on average similar to the home-based location service. Simulations of the hierarchical location service, including the optimizations of Chapter 4, with real world data on mobility and replication patterns will have to determine which approach will provide better performance.

3.5 Implementing Physical Node Selection

The location service is implemented using physical nodes that are grouped together to form logical nodes that, in turn, form the distributed search tree of the location service. The design of a physical node can be divided into three layers: the algorithm, distribution,

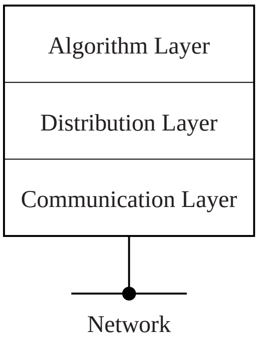


Figure 3.16: Layering in the design of a physical node.

and communication layer (see Figure 3.16). The algorithm layer contains the implementation of the update and lookup procedures, as described in Chapter 2. The procedures use only the logical search tree and have no knowledge of node partitioning. The distribution layer provides an RPC interface to the algorithm layer. It contains, however, only the code responsible for selecting the physical node that stores the contact record of an object. The communication layer is responsible for the actual communication.

The algorithm layer uses **logical node identifiers** to identify nodes in the search tree, such as the parent. The distribution layer takes a logical node identifier and an object handle and converts those to the **physical node identifier** that identifies the physical node that stores the contact record for the object identified by the object handle. The communication layer implements the RPC semantics by exchanging messages with the physical node selected by the distribution layer. The communication layer is responsible for resolving the physical node identifier to the network address of the physical node.

The distribution layer works conceptually as follows. When a procedure in the algorithm layer needs to communicate with a certain logical node, for instance, the parent, the procedure invokes the RPC primitive provided by the distribution layer. The distribution layer computes for every physical node of the logical parent node the distance between the location of the physical node and the location in the selection field of the object handle. When the distances to all physical nodes are known, the distribution layer selects the physical node with the shortest distance. If multiple physical nodes exist at the same location, the random field is used to choose which physical node to use. The distribution layer subsequently initiates a message exchange by invoking the RPC algorithms in the communication layer. To compute the distances, the distribution layer needs to know the geographical location of all the physical nodes of the logical nodes with which it will communicate, such as, the parent and child nodes.

3.5.1 Requirements

To come to an efficient implementation, the location-aware selection method has to fulfill the following requirements.

R1 *The selection process should be deterministic and unique.*
As long as the search tree does not change, the same physical node should be selected. Moreover, since it is inefficient to have to check multiple physical nodes, we should avoid ambiguity and select only one physical node.

R2 *The location information should occupy a small number of bits in the object handle.*
Since object handles are used as general object references in our system, a large number of them will be used. Object handles are therefore not allowed to become too big.

R3 *The selection process should be fast.*
Since the physical node selection process is on the critical path for communication, it should have a small overhead, that is, take as little time as possible.

R4 *It should be easy to add, remove, or move physical nodes.*
Since we can imagine the logical tree and its partitioning being adapted regularly to suit the current situation, these modifications should not require much work or have a large impact on the search tree as a whole.

R5 *The storage overhead introduced by partitioning and selecting a physical node should be reasonable.*

R6 *The communication overhead in both bandwidth and latency, should also be reasonable.*

Requirements R5 and R6, basically state that the extra storage and communication overhead should not endanger the usability of the location service.

3.5.2 General Implementation

In principle, we can recompute the distances for the same or similar locations every time a particular geographical location is used in communication. However, if we consider that the logical root node might have between 10^3 and 10^5 physical nodes, computing all distances every time is clearly undesirable, given requirement R3. We can, however, take the distance computation out of the critical communication path, by creating a **location-mapping table** off-line and using the location as an index in this table.

We create the location-mapping table, as follows. We divide the surface of the earth into a large number of small disjoint elementary areas. This division is independent of the partitioning used by the search tree. Consider a logical node N that has been partitioned into k physical nodes $PN_1, \ldots, PN_k$. We assign physical node PN_i to elementary area A if PN_i is the closest physical node to A. Each tuple (A, PN_i) forms an entry in the mapping

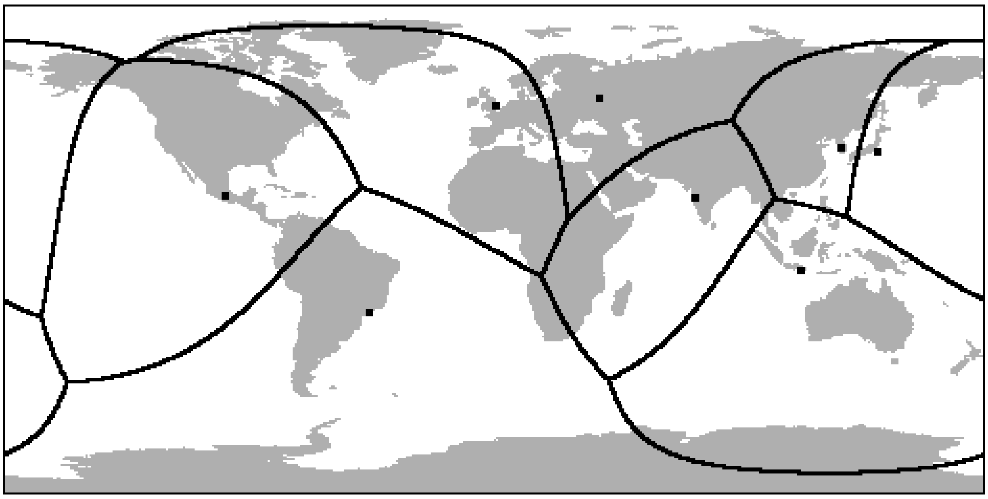


Figure 3.17: The mapping table for the world domain with a logical root node consisting of eight physical nodes placed in Surabaya, Sao Paolo, Seoul, Mexico City, Moscow, Tokyo, Bombay, and London.

table of logical node N. The mapping table of node N is distributed to all physical nodes that may need to communicate with node N. When a physical node is added to or removed from the set of physical nodes, the mapping table of node N needs to be recomputed and distributed again. This recomputation and distribution is described in detail in Chapter 7.

Figure 3.17 shows, as an example of a mapping table, the mapping table of the root node. The figure shows the mapping table projected on top of a map of the world. In this example the root node consists of eight physical nodes placed at eight major cities in the world. In the figure the eight cities are represented as dots. The area surrounding a city consists of all the elementary areas that will be mapped to that particular city. The areas surrounding the cities are distorted because the distances are computed using a sphere to approximate the surface of the Earth and the figure shows the Mercator projection of the Earth onto a flat surface.

3.5.3 An Array-based Implementation

A straightforward way to create elementary areas on the surface of the earth is by laying a grid on the surface using longitude and latitude. The longitude ranges from 180° west to 180° east of Greenwich and the latitude ranges from 90° north to 90° south of the equator. If we use, for example, $1^\circ \times 1^\circ$ degree areas, this results in 64,800 elementary areas for the complete surface of the earth. We implement the mapping table using a 2-dimensional array, using the (longitude,latitude) coordinates of an elementary area as an index in the array.

This implementation fulfills most requirements easily. The mapping table ensures that the selection process is deterministic and unique, fulfilling requirement R1. The second requirement (size of location information) depends heavily on the resolution (size of an elementary area) used. In the example above the size is at least 17 bits. Requirement R3 (fast execution) is fulfilled by using an efficient table-indexing operation. Since adding or removing a physical node simply requires recomputing and redistributing the mapping table, requirement R4 is easily met. Adding or removing a physical node also requires the redistribution of the contact records over the physical nodes. This redistribution process is described in Chapter 7.

Meeting requirement R5 and R6 (small storage and communication overhead) depends, just like R3, heavily on the resolution used. If we use a 4-byte physical node identifier, the example above gives mapping tables with a size of $64,800 \times 4 = 253$ kilobytes. Given that a $1^\circ \times 1^\circ$ elementary area is at most in the order of 100 km × 100 km, a mapping table of this resolution will provide a resolution in line with our assumption that geographical distances provides a good metric at distances over 1,000 km.

3.6 Location Awareness Revisited

In retrospect, there is reason to reconsider our geography-based load distribution scheme. The main goal of the location-aware load-distribution scheme is to provide a shorter communication distance than the hashing load-distribution scheme. Unfortunately, under the examined object models, the scheme results in an improvement of only 20% of the geographical distance traveled. The main difficulty with this improvement is that, while it is a reasonable improvement for the geographical distance, it is unclear how much of that improvement will find its way into the actual network distance.

This problem is compounded by the results of a small experiment we performed after developing the location-aware load-distribution scheme. In the experiment, fully described in [Ballintijn et al., 2000], we examined the network and geographical distances between 19 cities distributed across the world. When we compared the network and geographical distance between pairs of cities, we found that there is currently no correlation between their geographical and network distance in the Internet.

This basically means that the location-aware load-distribution scheme cannot be applied in the current Internet. We still expect the network distance to become more proportional to the geographical distance in the future. This expectation is built on the fact that within the USA and Europe network distances show significantly more correlation with geographical distance than on a worldwide scale. We could also try to find another distance metric that has a correlation with network distance and still has the same characteristics as the geographical distance (i.e., is a metric space), but whether such a metric actually exists is unknown. Fortunately, we can still use the hashing load-distribution scheme to avoid the workload scalability problem in the root node.

Chapter 4

An Efficient Lookup Operation

This chapter looks at two separate methods for improving the efficiency of the lookup operation. The first method forms the main part of this chapter. It consists of a caching technique that decreases the load in the location service and shortens the duration of the lookup operation. The second method is discussed in the final section of this chapter. It is an extension to our basic search tree that allows us to efficiently distinguish different types of replicas during a lookup operation.

The focus in this chapter is on traversal in the logical search tree only. The physical search tree, as described in the previous chapter, plays no part since node partitioning is completely transparent to lookup operations. To ease our discussion, we assume in this chapter that mobile objects are never replicated. A mobile object is thus always present at only a single location.

4.1 Preventing Tree Traversal

In this section we describe the efficiency problem of using general tree traversal to search for contact addresses in the location service tree, and provide a caching method to improve efficiency.

4.1.1 Problem

The simplified lookup operation, as described in Chapter 2, is inefficient when dealing with nonlocal contact addresses. To retrieve these kinds of addresses, the operation needs to travel up to the root node and down again. This kind of general tree traversal is undesirable for two reasons. First, a general tree traversal visits a large number of nodes, increasing the workload at every node. This load increase is particularly apparent at higher level nodes, and endangers the *scalability* of the location service. Second, visiting a large number of nodes also results in the simplified lookup operation taking a long time to complete. Long delays endanger the *usefulness* of the location service since no client wants to wait a long time to get a contact address to actually start using an object.

General tree traversal is, however, unavoidable when dealing with unpopular objects that are located far away. What is more problematic is that when two lookup operations for the same nonlocal object are started in the same leaf domain both lookup operations will perform the same costly tree traversal. We clearly would like to avoid the same tree traversal in the second lookup operation. The usual way to avoid performing the same operation twice is to store the result of a previous operation (a contact address in our case) in a cache and reuse the result when needed. This approach is described in the context of wide-area naming systems in [Lampson, 1986; Cheriton and Mann, 1989].

Unfortunately, caching contact addresses is ineffective when dealing with *mobile* objects. Caching contact addresses would require a stable mapping from object handle to contact address, but having a stable mapping is impossible since mobile objects frequently change location and thus contact address. Every time a mobile object moves, it inserts the contact address of its new location in the location service and deletes the contact address of the old location. Since the results stored in a cache become invalid when the object moves, the reuse of the results stored in the contact address caches is precluded. A similar problem occurs when contact addresses would be replicated at other tree nodes, which is proposed in [Jannink et al., 1997].

A commonly proposed solution is to cache contact addresses only when they are stable [Pitoura and Samaras, 2001]. This situation is indicated by a high lookup-to-mobility ratio. Unfortunately, this solution is not a good one since it ensures only that contact addresses for highly mobile object are not cached. It does not provide an alternative method to prevent tree traversal in the case of highly mobile objects. Fortunately, a different view on the mobility problem allows us to take a different approach, resulting in a caching technique called **location caching**.

4.1.2 Location Caching

The general principle behind every caching scheme is that by saving a value that required a significant amount of work, we can avoid doing that work a second time when the value is needed some time in the future. To be useful in the future, the value saved in the cache must be stable. The instability of contact addresses made them unsuitable for caching. We can, however, search for different kinds of values that are stable and also allow us to avoid traversing the search tree.

To arrive at such a stable value, we introduce the **mobility domain**. The mobility domain of a mobile object is the smallest domain in which the object is continuously present. For instance, if a mobile object divides its time evenly over Amsterdam, London, and Paris, its mobility domain would be Europe. Even though the object is mobile, its presence in the European domain is stable.

The directory node associated with the mobility domain always stores a contact record for the object. The contents of that contact record, however, changes every time the object moves. In our example, the `Europe` node always stores a contact record for our mobile object, and depending on whether the object is currently located in Amsterdam, London, or Paris, the contact record stores a forwarding pointer to the `Netherlands`, `UK`, or `France` node, respectively.

A reference to a stable node, such as the `Europe` node, is a stable value that we can use in a caching scheme. To introduce caching in the location service, we add **location caches** to all the directory nodes in the search tree. In these location caches we save a reference to the stable node of an object. We can now shorten the tree traversal of the lookup operation by using the cached references to continue the lookup operation directly at the stable node. We thereby avoid visiting tree nodes located higher in the tree when they have been visited recently.

We shorten the tree traversal even further by storing the current contact address of an object at the stable node itself instead of at the leaf node where the object currently resides. The lookup operation can now retrieve the contact address directly at the stable node instead of having to travel downward to the leaf node. We call the stable directory node the **stable address location** for the mobile object since the node is the location where the current contact address for the mobile object is always present. A location cache thus stores references to stable address locations.

Figure 4.1 shows a part of the search tree for an example mobile object. The mobility domain of the mobile object is the UK domain and its stable address location is thus the `UK` node. The current contact address of the mobile object is therefore stored at the `UK` node, even though the object currently resides in the Glasgow domain. In the example, the location caches at the `Amsterdam` and `Netherlands` nodes store, for this mobile object, references to its stable address location. A lookup operation started at the `Amsterdam` node would use the reference to directly retrieve the current contact address at the `UK` node.

In the optimal case, the lookup operation consists of two steps, as shown by the example. First, the lookup operation will visit the local leaf node, which stores a reference in its cache. Second, using the reference, the lookup operation will visit the stable address location to retrieve the current contact address. Moreover, lookup operations started at adjacent leaf nodes are also optimized. For example, in Figure 4.1, we can see that a lookup operation initiated at the `Rotterdam` node is also improved. The operation finds a cached node reference at the parent of the `Rotterdam` node (i.e., the `Netherlands` node), and avoids visiting the `Europe` node.

The location caches are filled as a side effect of the lookup operation. We extend the lookup operation to return, besides a contact address, also a reference to the node where that contact address was found. The lookup operation stores this reference in the cache of every node it has visited. For instance, in our example, the cached node reference at the `Netherlands` and `Amsterdam` nodes were installed by a previous lookup operation started at the `Amsterdam` node. To reference nodes, we simply use our normal (logical) node identifier.

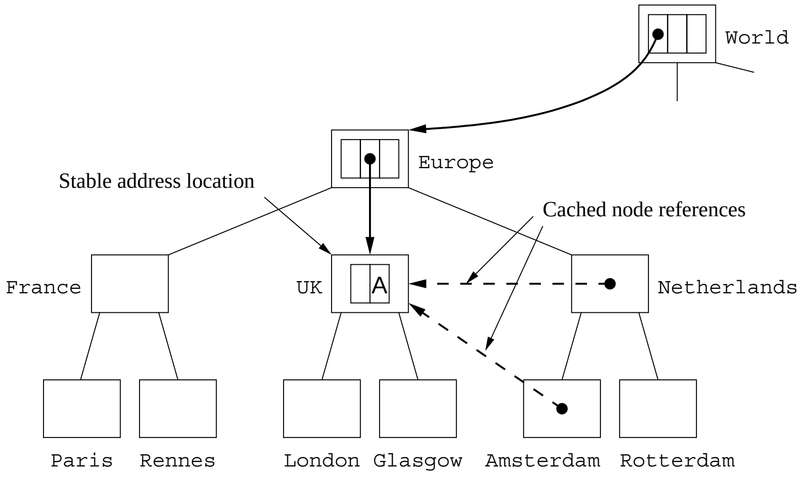


Figure 4.1: Cached location pointers at the `Amsterdam` and `Netherlands` nodes point to the stable address location of the object, in this case, the `UK` node.

4.2 Stable Address Location Management

For location caching to be effective, the contact address of a mobile object must be stored at its stable address location. In this section we describe the methods to achieve this requirement. We assume for the rest of this chapter that the movement of a mobile object is visible in the location service as an insert operation followed by a delete operation, in that order. The mobile object thus remains accessible at its old location during its change of location.

4.2.1 Stable Address Location Identification

In the rest of this chapter we use the following, more formal definition of the term stable address location. The stable address location for a mobile object is the lowest directory node in the search tree where a contact record for the object is continuously present. Since the object is mobile, we expect the contents of this contact record to change frequently. New contact addresses will be inserted in the contact record and old contact addresses will be deleted. Using this definition, the stable address location for nonmobile objects becomes the leaf node, as desired.

We specifically use the lowest directory node in the search tree that corresponds to the smallest domain where the object is mobile. Even though directory nodes higher up in the tree (i.e., enclosing domains, such as the root domain) would also provide a stable node, it is undesirable to choose the high-level nodes as the stable address location since it prevents the locality of the lookup and update operation.

Unfortunately, we cannot determine in advance whether an object is mobile, and if so, what its stable address location is. We need to determine this dynamically during the lifetime of the object. We therefore initially assume that an object is located at a fixed location, and gather mobility information on the object while it exists. The mobility domain and thus the stable address location of a mobile object will become apparent when it is moving around. When the stable address location is known, we can store the contact address of the mobile object there, instead of in a leaf node.

What is more, a mobile object may change its mobility behavior and thus have several different stable address locations during its lifetime. We therefore need to continue with gathering mobility information (even after a stable address location is found) and move the contact address up or down the tree when needed. For instance, in our example in Figure 4.1, the current stable address location is the `UK` node. However, if the mobile object were to start visiting Amsterdam and Rennes regularly as well, the `Europe` node would become the stable address location for the object and current address of the object would be stored at the `Europe` node.

Two aspects of an object's mobility behavior determine its stable address location. The first aspect is the frequency with which update operations occur for an object, that is, the frequency with which the object moves. If this frequency is too low, an object stays at the same location for a long time and it is not worthwhile to store contact addresses of this object higher in the search tree. The second aspect is the area from which the update operations originate, that is, the area in which the object is mobile. When this area becomes larger, the contact address of the object needs to be stored higher in the tree.

To determine the update frequency and area, every directory node collects per-object historical update information. The update information records the frequency of update requests arriving at a directory node. The directory node summarizes this information in a single value per object, indicated by H_{now}. We call this value the **update history** of a contact record. The update history is basically a weighted average of the time between update requests for an object at a node. Intuitively, it keeps track of how long an object usually remains in one of the subdomains of the directory node based on recent moves. A high value for H_{now} means that the object has been in a single subdomain for long periods of time. A low value means that the object has been frequently moving from one subdomain to the next.

To compute the update history of a contact record, we maintain for every contact field f in the contact record the last time it was filled, indicated by $T_{filled}(f)$. This value is the last time that contact field f went from being empty to being filled with a contact address or forwarding pointer. The last filled time is undefined when the contact field is empty. Adding an extra contact address to a contact field does not change the last filled time of the contact field since it was already filled. To indicate that a contact field is not empty and its last filled time is defined, we use the following notation: $T_{filled}(f) \neq \perp$.

The update history is computed in the following way:

$$
\begin{aligned}
D &:= T_{now} - \max\{T_{filled}(f) \mid f \in \mathtt{CR} \land T_{filled}(f) \neq \bot\} \\
H_{now} &:= \alpha \cdot D + (1-\alpha) \cdot H_{prev}
\end{aligned}
$$

The formula consists of two steps. First, we compute duration D, which indicates how much time has passed since the most recent time a contact field in the contact record was filled. A recently inserted contact address results in a low value for D. Second, we use this duration D and the previous history value H_{prev} to compute the current history value H_{now}. A low value for H_{now} means frequent updates on the contact record have occured in the recent past. The aging factor α (with $0 < \alpha < 1$) determines the influence of historical data on the current update history. A large α weights the current change heavily; a small one weights previous changes more heavily.

When a new contact record is created, we need to give its associated update history an initial value. Since the object is initially assumed to be nonmobile, we give the update history a large value. When a contact record is deleted, its history value is also deleted. Since a contact record is deleted when its object leaves a domain, the update history of a mobile object is effectively reset to the initial value every time the object leaves and enters a domain. The contact record of a mobile object that regularly enters and leaves a particular domain therefore has a high history value at the associated directory node.

Now we have the situation that the update history has a low value only at the stable address location. At nodes higher in tree the value is large because no new insert procedures are executed. All insert operations originate from the mobility domain, visit the stable address location, and effectively finalize their execution there, lowering the update history. At nodes lower in the tree the update history is also large. At these nodes the update history is large because the value is reset every time the object moves out of the associated smaller domains.

Figure 4.2 shows the update history values for an object that is mobile in the UK domain. The contact records of the mobile object at the `World` and `Europe` nodes were created when the first contact address of the mobile object was inserted and the nodes needed to store forwarding pointers. However, since the object moves within the UK domain only, the resulting insert and delete operations never change the contact records at the `World` and `Europe` node. These nodes therefore associate a high update history value with their contact records.

The contact records at the `London` and `Glasgow` nodes also have a high update history value, but in this case, it is because the mobile object is never located long enough in their respective domains. Since the contact records at the `London` and `Glasgow` nodes start with a high initial value and the object is only in their domains for a short time, there is no possibility for their update history to get smaller. The only node that continuously sees all insert operations is the `UK` node. It therefore has a low update history value.

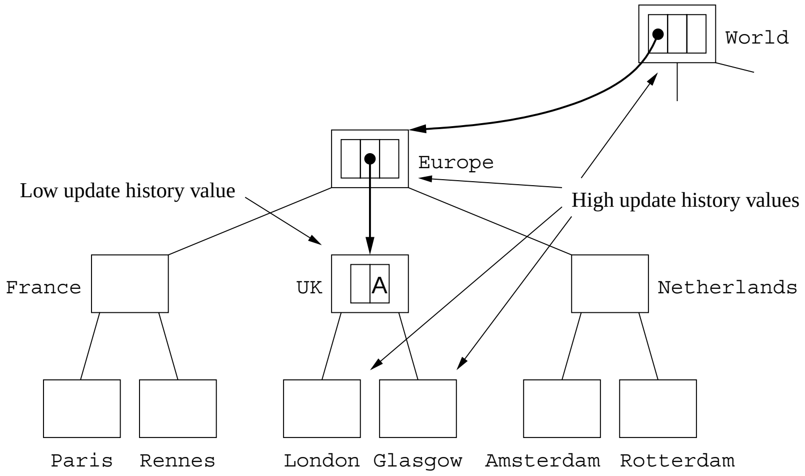


Figure 4.2: The update history values for an object mobile in the UK domain. Only the value at the UK node is low.

4.2.2 Moving the Stable Address Location Upward

When the location service determines that the stable address location of a mobile object is located at a higher node than the node that currently stores the contact address, it needs to move the address up the tree to this new directory node. Since we are dealing with a mobile object, we know new contact addresses are regularly inserted in the location service. We can therefore use the normal insert operation to "move" the current contact address to a node located higher in the tree. The next time the object changes its contact address, we simply store its new contact address at the new stable address location higher in the search tree.

As described in Chapter 2, the insert operation uses a distributed decision making process to determine at what directory node on the path from leaf to root node to store a contact address. In the process, every directory node first determines for itself whether it can and wants to store the contact address. The node then asks permission from its parent to store the contact address itself or at one of its children. The parent can agree with the request or decide it wants to store the contact address itself. The highest node in the tree that wants to store the contact address actually stores the address.

Every directory node involved in the distributed decision making process uses the *doStoreHere* function to determine whether it wants to store the contact address itself. The *doStoreHere* function returns a boolean value, with *true* indicating that the contact address should be stored at the local directory node. All we need to do is make sure that the new stable address location is the highest directory node where the *doStoreHere* function returns *true*. The insert operation using the distributed decision process will then store the new contact address at the stable address location.

We let the return value of the *doStoreHere* function depend on the update history of the contact record of the mobile object and the **mobility threshold**, indicated by $D_{mobility}$. The *doStoreHere* function basically takes the following form:

$$H_{now} < D_{mobility}$$

During the insert operation, every directory node uses this inequality to determine whether it is better suited than its child node to store the contact address. When an object is unknown, there is no update history for the *doStoreHere* function to use. To ensure that a contact address is stored at a leaf node (the default situation) in this case, we do two things. First, we let the *doStoreHere* function always returns *true* at a leaf node. Second, we let the *doStoreHere* function return *false* at a nonleaf node.

4.2.3 Moving the Stable Address Location Downward

The stable address location moves down the tree when the object decreases its mobility domain (i.e., starts moving in a smaller area). When this happens the contact address of the object should be moved down the tree to the new stable address location. For instance, when the object in Figure 4.1, which was mobile in the UK domain, becomes stable in the Glasgow domain, its contact address should be moved from the `UK` node to the `Glasgow` node. Moving the address downward in the search tree improves the locality of future update operations and lookup operations from the domain in which the object is mobile.

Old cache entries, such as those pointing to the `UK` node, remain valid and useful after an address is moved down. Since the `UK` node stores a forwarding pointer to the `Glasgow` node, a cache reference to the `UK` node can still be used to quickly find a contact address. Since a reference to the `UK` node is not as efficient as a reference to the `Glasgow` node itself, it will be updated on the next lookup operation.

A node uses the **stability threshold**, indicated by $D_{stability}$, to decide whether its child node has become more suited to store a contact address that it currently stores itself. The stability threshold is used in the following equation:

$$T_{now} - T_{filled}(f) > D_{stability}$$

A directory node uses the inequality to decide whether the last time the contact field was filled with a contact address (after being empty) was a long time ago. If so, the object has apparently a stable presence in the child domain associated with the contact field, and the child node has thus become a more appropriate node to store the contact address. Note that the equation applies only to a single contact field that stores a contact address, and not the contact record as a whole.

We are only interested in the last time the contact field was filled; we are not interested in whether the contents of the contact field changed. Consider the situation where the contents of a contact field changes frequently but there is always at least one contact address present. In this situation, the mobile object is moving inside the child domain associated with the contact field but it is always present. The child domain is therefore the mobility domain since it is the smallest domain in which the object is always present.

Unfortunately, we cannot use the normal insert operation to store a new contact address at the new (lower) stable address location. Since the old stable address location still holds a contact address, our consistency rules (as described in Chapter 2) require the new contact addresses to be stored there as well. We therefore need a separate move-down operation. Since this operation does not need to be invoked by clients, it is internal to the location service. The move-down operation is implemented using an individual **take-over message** and the *reInsertAddress* and *reInsertPointer* procedures.

The move-down operation starts when the parent node decides the contact address for a particular object needs to be stored at one of its child nodes. The parent sends the take-over message to the child node specifying the object and contact address concerned. The child node executes the *reInsertAddress* procedure to insert the contact address. To finish the move-down operation the child node invokes the *reInsertPointer* procedure at the parent node. This procedure swaps the contact address for a forwarding pointer, and the move-down operation has finished.

The move-down operation uses only a single message instead of an RPC when the parent node requests the *reInsertAddress* procedure at its child node. We avoid the more heavyweight RPC since the parent node is not interested in knowing when the move-down operation is completed. When the parent node detects a favorable situation for a move-down operation, it simply send the take-over message. After sending it, the parent can forget that it has sent the message, knowing that in the normal case its child node will continue the move-down operation. If the move-down operation needed to be aborted, for instance, because of concurrent update operations, the parent node will detect the favorable situation again in the future, if it still exists, and try a second time. The use of the lightweight take-over message is an application of our idempotency principle. We do not have to worry about the take-over message getting lost. Since the favorable situation will persists in that case, the parent will simply decide at a later time to send the message again.

Figures 4.3, 4.4, and 4.5 show the execution of the move-down operation in our example search tree. In Figure 4.3, the object is currently located in the Glasgow domain, and its contact address is stored at the `UK` node. The object was previously mobile in the UK domain, but using the stability threshold the `UK` node has determined that the object has become stable within the Glasgow domain, and that the contact address should be stored at the `Glasgow` node. The `UK` node therefore initiates the move-down operation by sending the take-over message to the `Glasgow` node, in step 1. Upon receiving this message, the `Glasgow` node tentatively inserts the contact address.

Figure 4.4 shows step 2 of the move-down operation. As part of the *reinsertAddress* procedure, the `Glasgow` node invokes the *reInsertPointer* procedure at its parent, the `UK` node. The `UK` node uses the *reInsertPointer* procedure to exchange the contact address with a forwarding pointer. Figure 4.5 shows the third and final step of the move-down operation. The `UK` node signals the `Glasgow` node that the *reInsertPointer* procedure has successfully completed. Upon receiving this reply, the `Glasgow` node knows the

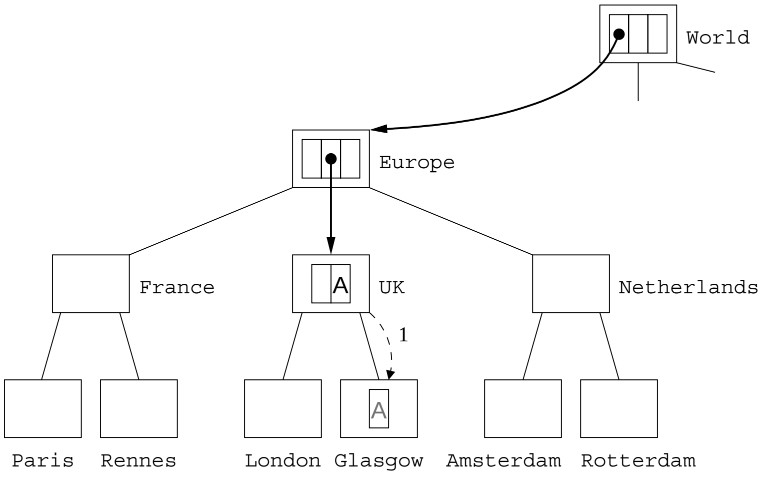


Figure 4.3: Moving down a contact address. The `UK` node initiates the move-down operation by sending a take-over message to the `Glasgow` node.

move-down operation was successful, and the tentative contact address is permanently stored at the node.

There are two reasons for choosing the current structure for the move-down operation, that is, inserting the contact address at the child node before exchanging the contact address for a forwarding pointer at the parent node. The first reason is that in the chosen order the contact address stays available for concurrently executing lookup operations. The second reason has to do with crash recovery and concurrent update operations, and is dealt with in Chapter 5.

The move-down operation is carefully constructed to avoid a race condition during its execution. The race condition could occur when a move-down operation and a delete operation for the same contact address execute concurrently. The method to avoid this race condition is explained in Figure 4.6. In this figure, there is a parent node `N`, a child node `CN`, and an object. The parent node `N` decides that it wants to move the contact address of the object to its child node `CN`. Shortly afterward, the object deletes its contact address.

In step 1 in Figure 4.6, parent node `N` sends a take-over message to child node `CN`. In step 2, the object sends its delete request to child node `CN`. The race condition occurs when the delete requests arrives at child node `CN` before the take-over message. Since the contact address is (still) stored at parent node `N`, the child node simply forwards the delete request to the parent, in step 3. This request crosses the take-over message. Now the take-over message arrives at child node `CN`, and the child inserts the contact address. In the meantime, the delete request arrives at parent node `N`, and the parent deletes the contact address. However, in step 4, child node `CN` sends a *reInsertPointer* request to insert a

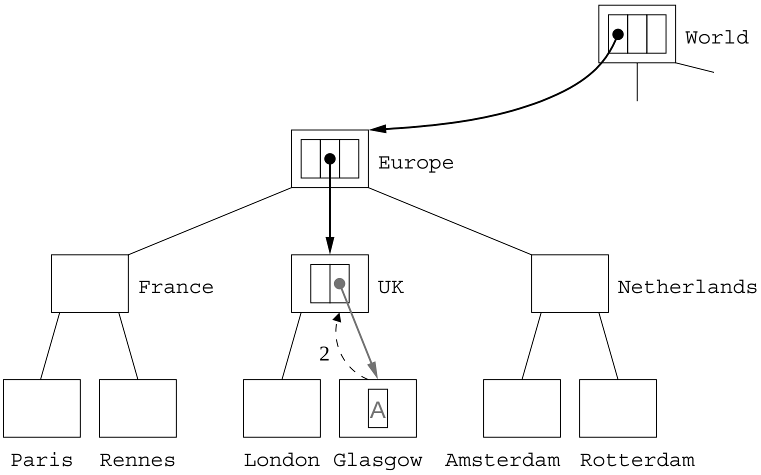


Figure 4.4: Moving down a contact address. The `Glasgow` node invokes the *reInsertPointer* procedure at the `UK` node.

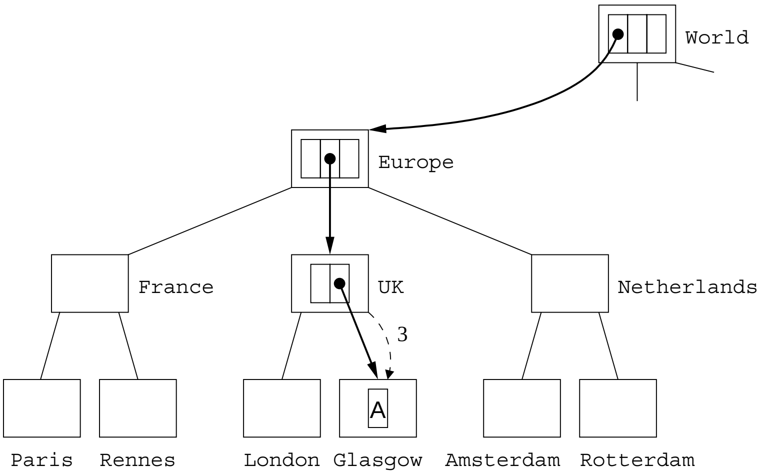


Figure 4.5: Moving down a contact address. The `UK` node sends an RPC reply to confirm the successful exchange of the contact address with a forwarding pointer.

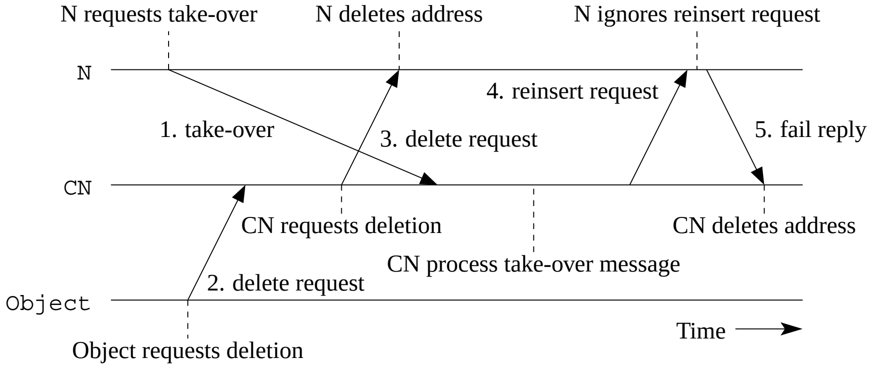


Figure 4.6: Solution to the race condition between the move-down and delete operation.

forwarding pointer at parent node N. If the parent node would consider only this request and simply insert a forwarding pointer, the end result of this scenario would be that the delete operation is ignored since both the contact address and the forwarding pointer will be present. To avoid this race condition, the *reInsertPointer* procedure needs to check that the contact address is actually still present at the parent node. If the address is not present at the parent, the procedure *reInsertPointer* should fail, change nothing, and return a fail reply, as in step 5. Furthermore, child node CN should delete the contact address when it receives the fail reply.

A concurrent insert operation can also create problems for the move-down operation. These problems can occur both in the *reInsertAddress* procedure at the child node and in the *reInsertPointer* procedure at the parent. When a concurrent insert operation inserts a forwarding pointer in the contact field to be used by the move-down operation at the child node, the *reInsertAddress* procedure must fail since inserting an address in a contact field with a forwarding pointer violates our consistency requirements. Likewise, when a concurrent insert operation inserts a new contact address in the contact field to be used by the move-down operation at the parent node, the *reInsertPointer* procedure must fail. If there is another contact address present in the contact field, the specified contact address cannot swapped for a forwarding pointer without violating consistency rule C3, described in Chapter 2. In both cases, the move-down operation is simply aborted and retried at a later time. Concurrency issues are further discussed in Chapter 5, when we consider nodes that might fail.

The reInsertAddress Procedure

Listing 4.1 shows the code of the *reInsertAddress* procedure. The reinsert operation is invoked when a take-over message is received from the parent. This is not a normal RPC invocation since the parent node is not waiting for a reply message. The standard RPC

```
(1)  procedure reInsertAddress(oh : ObjectHandle,
(2)                            addr : ContactAddress) is
(3)    origCR : ContactRecord;
(4)    child : NodeID;
(5)    success : Boolean;
(6)  begin
(7)    -- Retrieve a copy of the contact record, or create one, if needed
(8)    origCR := crDatabase(oh);
(9)    if origCR = NIL then
(10)     origCR := new ContactRecord;
(11)     initializeCR(origCR, children, thisNode);
(12)   end if;
(13)   child := determineField(addr);
(14)   -- Determine if we can and need to insert the contact address
(15)   if origCR(child).isPtr or addr ∈ origCR(child).addrSet then
(16)     return;
(17)   end if;
(18)   crDatabase(oh)(child).addrSet := origCR(child).addrSet + {addr};
(19)   success := call reInsertPointer(oh, addr, thisNode) at parent;
(20)   -- Undo the local modification, if the address is deleted at the parent
(21)   if not success then
(22)     if isEmpty(origCR)
(23)       then delete crDatabase(oh);                 -- Remove temporary record
(24)        else crDatabase(oh) := origCR;              -- Restore original record
(25)     end if;
(26)   end if;
(27) end reInsertAddress;
```

Listing 4.1: The *reInsertAddress* procedure.

parameter *caller* is therefore also not present. The take-over message does contain an object handle and contact address, and these are given to the procedure as parameters, as shown in lines 1–2.

The procedure uses three local variables, as shown in lines 3–5. The contact record *origCR* contains the original contents of the contact record in the contact record database. The node identifier *child* is used to indicate the specific contact field the contact address should be stored in. The boolean variable *success* stores the reply of the parent node, indicating whether the parent node successfully exchanged the contact address for a forwarding pointer.

The *reInsertAddress* procedure starts by retrieving a copy of the contact record from the database using the object handle, in line 8. If the object is unknown in this directory node, which is the normal case, there is no contact record in the contact record database (line 9) and a new contact record needs to be created and initialized, in lines 10–11. However, given the possibility of concurrent insert operations, a contact record might already exist.

The next thing the procedure does, in line 13, is determine at which subdomain the contact address is located. This should be known since it determines which contact field of the contact record to use. To determine this subdomain, we store in each contact address the leaf domain where the address resides. Using its knowledge of domain hierarchy, the directory node can determine to which of its subdomains the leaf domain belongs. The subdomain is stored in the variable *child*.

The *reInsertAddress* procedure then determines, in line 15, whether it can and needs to insert the contact address in the contact record. First, it determines whether it actually can insert the address. The only reason why it cannot insert a contact address is that the contact field already contains a forwarding pointer. Such a forwarding pointer could have been inserted by a concurrent insert operation or a subsequent move-down operation. Either way, the move-down operation fails, and the procedure is finished, in line 16.

Second, the *reInsertAddress* procedure determines whether it actually needs to insert the address. If the address was already moved down before, but the parent did not know about this yet, the procedure is already done. Since the parent node does not record that is has sent a take-over message, it might actually send two take-over message shortly after each other. In this case, there is nothing to be done, and the procedure is finished, in line 16.

The procedure can now safely insert the contact address in the contact record, in line 18. It then sends the *reInsertPointer* request to the parent node, and awaits its reply, in line 19. The reply value *success* describes whether the contact address was successfully swapped for a forwarding pointer at the parent node. If that is the case, the search tree is consistent again and the move-down operation has finished successfully.

If the parent node did not successfully exchange a contact address for a forwarding pointer, the move-down operation is aborted and the local changes need to be undone, in lines 21–26. If the contact record was empty before the address was inserted, the record is deleted from the contact record database, in line 23, otherwise the original contact record is restored, in line 24.

The reInsertPointer Procedure

Listing 4.2 shows the code of the *reInsertPointer* procedure. Its task is to exchange a contact address with a forwarding pointer. The *reInsertPointer* procedure takes three parameters, as shown in lines 1–3. Object handle *oh* specifies for which object the contact address should be exchanged, and contact address *addr* specifies the contact address that should be exchanged. The last parameter is the standard *caller* node identifier since the *reInsertPointer* procedure is invoked using the normal RPC mechanism.

The procedure starts by checking whether a contact record exists for the object, in line 5. If no contact record exists, there was apparently a concurrent delete operation, as discussed above. The move-down operation is not needed anymore, and the procedure returns *false*, in line 6, to tell the child node to undo its modification.

If a contact record exists, the *reInsertPointer* procedure checks whether the contact

```
(1)  procedure reInsertPointer($oh$ : ObjectHandle,
(2)                            $addr$ : ContactAddress,
(3)                            $caller$ : NodeID) return Boolean is
(4)  begin
(5)    if $crDatabase(oh) = NIL$ then
(6)      return false;
(7)    elsif $crDatabase(oh)(caller).addrSet = \{addr\}$ then
(8)      $crDatabase(oh)(caller).addrSet := \emptyset$;
(9)      $crDatabase(oh)(caller).isPtr := true$;
(10)     return true;
(11)   elsif $crDatabase(oh)(caller).isPtr$ then
(12)     return true;
(13)   else
(14)     return false;
(15)   end if;
(16) end reInsertPointer;
```

Listing 4.2: The *reInsertPointer* procedure.

field for the caller (i.e., the child node) stores only the contact address that should be moved down, in line 7. If that is the case, the contact address can be safely deleted, in line 8, and a forwarding pointer can be inserted in its place, in line 9. The procedure has successfully exchanged the address, and returns *true*, in line 10, to inform the child node of this.

If the contact field stores a forwarding pointer (line 11), the procedure assumes the contact address was already moved down, and the procedure returns successfully as well, in line 12. Otherwise, the situation has apparently changed since the parent initiated the move-down operation, and the contact address can no longer be exchanged for a forwarding pointer. To indicate the unsuccessful execution, the procedure returns *false* to its child node, in line 14.

4.3 Location-cache Management

To implement our location caching and stable address location management, we need to introduce new data structures and global variables. As indicated in the beginning, to implement location caching the lookup operation needs to return more than simply a contact address. It also needs to return the node identifier of the directory node that stored the contact address. These two values are combined in the *StoredAddress* type, with the *addr* and *node* fields, respectively, as shown in Listing 4.3. A third field, *age*, is also added. This field stores the time elapsed since the contact address was originally stored at the directory node. This field is used to compute how long the reference to the node should be stored in the location cache.

The cache entry type *CacheEntry* is defined in Listing 4.4. An entry in the location

```
-- Value returned by the new lookupAddress procedure
type StoredAddress is
    record
        addr : Address := NIL;                    -- Contact address found
        age : Date := 0;                          -- Elapsed time between insert and lookup
        node : NodeID := NIL;                     -- Node where the address was stored
    end record;
```

Listing 4.3: The *StoredAddress* return type.

```
(1)  -- Per-object entry in location cache
(2)  type CacheEntry is
(3)      record
(4)          localPtrs : set of NodeID := ∅;                 -- Ptrs to nodes in current subdomain
(5)          remotePtrs : set of NodeID := ∅;                -- Ptrs to nodes outside current subdomain
(6)          expirationTimes : set (NodeID) of Date;         -- Expiration time for cache entry
(7)      end record;
(8)  -- Set of cache entries, indexed by object handle
(9)  type Cache is set (ObjectHandle) of CacheEntry;
(10) cache : Cache;
```

Listing 4.4: Data structure and global variable for the location cache.

cache consists of three field: *localPtrs*, *remotePtrs*, and *expirationTimes*. The first two fields are sets of node identifiers. In the cache, we distinguish between the node identifiers referring to directory nodes in the domain of the current node and the node identifiers referring to directory nodes in the rest of the search tree. This distinction allows us to support locality by using the identifiers in the *localPtrs* set before the identifiers in *remotePtrs* set in the lookup operation. The *expirationTimes* field is used to delete stale node references in the two sets. The expiration time of a node reference is based on the age of the contact address, found at the directory node and returned in the *StoredAddress* return value. The younger the contact address found, the sooner the node reference will expire. This approach resembles the Alex cache replacement policy that has been successfully applied to Web caches [Cate, 1992]. The location cache itself is defined as a global variable on line 10.

The *History* type is used to store the update history of each contact record stored at the current directory node. The history data structure is defined in line 2 in Listing 4.5. The history values are stored as an indexed set of floating-point values. We choose to keep the history value separate to keep our contact record data structure conceptually simple. In line 3, the type is used to define the global variable *history*.

To ease the handling of the location cache, we introduce the following three procedures: *cache_insert* (Listing 4.6), *cache_delete* (Listing 4.7), and *cache_lookup* (Listing 4.8). The *cache_insert* and *cache_delete* procedures are straightforward. Both proce-

```
-- Current update history for each contact record
type History is set (ObjectHandle) of float;
history : History;
```

Listing 4.5: Data structure and global variable for storing the update history.

```
(1)  procedure cache_insert(oh : ObjectHandle,
(2)                         node : NodeID,
(3)                         age : Date) is
(4)  begin
(5)    if node $\in$ domain(thisNode)                  -- Is the node inside our subtree?
(6)      then cache(oh).localPtrs := cache(oh).localPtrs + {node};
(7)       else cache(oh).remotePtrs := cache(oh).remotePtrs + {node};
(8)    end if;
(9)    cache(oh).expirationTimes(node) := expire(age);
(10) end cache_insert;
```

Listing 4.6: Inserting a node in the cache.

dures start by determining whether the node that is to be inserted or deleted is part of the local domain or is located outside it. Depending on the outcome, the node identifier is inserted in or deleted from the set of local or remote node references. The *cache_insert* procedure also sets the expiration time based on the age of the contact address, in line 9 of Listing 4.6. The *cache_delete* procedure checks whether the cache entry has become empty for the object, in line 7 of Listing 4.7, in which case the entry is deleted in line 8.

The *cache_lookup* procedure returns a node reference from either the set of local or remote node references, depending upon the *strategy* parameter. The use of the *strategy* parameter is explained in the next section on the new *lookupAddress* procedure. The procedure uses the procedure *nearest* to choose the directory node that is closest to the current directory node. The procedure *nearest* can use different metrics to determine which node

```
(1)  procedure cache_delete(oh : ObjectHandle, node : NodeID) is
(2)  begin
(3)    if node $\in$ domain(thisNode)                  -- Is the node inside our subtree?
(4)      then cache(oh).localPtrs := cache(oh).localPtrs − {node};
(5)       else cache(oh).remotePtrs := cache(oh).remotePtrs − {node};
(6)    end if;
(7)    if cache(oh).localPtrs = $\emptyset$ and cache(oh).localPtrs = $\emptyset$ then
(8)       delete cache(oh);                           -- Delete the whole cache entry.
(9)    end if;
(10) end cache_delete;
```

Listing 4.7: Deleting a node from the cache.

```
procedure cache_lookup(oh : ObjectHandle,
                       strategy : (local, remote)) return NodeID is
begin
  if strategy = local then               -- retrieve a node from the current domain
    return any in nearest(cache(oh).localPtrs, thisNode);
  else                                   -- retrieve a node from outside the current domain
    return any in nearest(cache(oh).remotePtrs, thisNode);
  end if;
end cache_lookup;
```

Listing 4.8: Looking up a node in the cache.

is closest to the current node, for example, geographical distance or the number of hops in the search tree. The use of the *nearest* procedure allows us to support locality by retrieving nearby contact addresses when those are available.

To simplify the new *lookupAddress* procedure, we introduce the *check_cache* procedure to retrieve a contact address using the location cache. The procedure is shown in Listing 4.9. The *check_cache* procedure takes as parameters the object handle for which a contact address should be retrieved and a strategy telling whether to use local or remotely located directory nodes. The procedure starts, in line 6, by checking whether the location cache has a reference to a directory node that potentially stores a contact address. If such a node exists (line 7), the procedure tries to retrieve it using an RPC invocation, in lines 8–9.

If a contact address was found using the cached node reference (line 10), the node reference of the contact address is inserted in the location cache, in line 11. If the node reference refers to the same node as the one found in the cache, this insertion is done to update the expiration time of the cache entry. The contact address can, however, also be found at a different node. In which case (line 12), the old node reference needs to be deleted, in line 13. If no contact address was found, the cache entry also needs to be deleted, in line 16. Either way, the result of the lookup request is returned to the *lookupAddress* procedure, in line 18. If no cache entry was found, the *NIL* value is returned to the *lookupAddress* procedure to indicate failure, in line 20.

4.4 The New Lookup Operation

The improved *lookupAddress* procedure that uses the location cache is shown in Listing 4.10 and Listing 4.11. The *lookupAddress* procedure takes three parameters and returns a value of the *StoredAddress* type, as shown in lines 1–3. The object handle parameter *oh* specifies the object for which we are looking for a contact address. The boolean parameter *subDomOnly* specifies whether the lookup procedure should keep to the current subdomain. The use of location caches introduces the risk of loops in the search tree that could conceivably allow the lookup procedure to follow cached node references endlessly and never terminate. The *subDomOnly* parameter limits the use of location caches to lo-

```
procedure check_cache(oh : ObjectHandle, strategy : (local, remote))
                                  return StoredAddress is
  storedAddr : StoredAddress;
  cachedNode : NodeID;
begin
  cachedNode := cache_lookup(oh, strategy);
  if cachedNode $\neq$ NIL then                         -- A cache entry was found.
    storedAddr := call lookupAddress(oh, true, thisNode)
                  at cachedNode;
    if storedAddr $\neq$ NIL then                     -- A contact address was found.
      cache_insert(oh, storedAddr);
      if storedAddr.node $\neq$ cachedNode then           -- The address was found
        cache_delete(oh, cachedNode);                  -- at a different node.
      end if;
    else                                       -- No contact address was found.
      cache_delete(oh, cachedNode);
    end if;
    return storedAddr;
  else                                               -- No cache entry was found.
    return NIL;
  end if;
end check_cache;
```

Listing 4.9: Using the cache to lookup a contact address at another node.

cal node references, and thereby ensures the termination of the lookup operation. The last parameter is the standard parameter *caller*, identifying the directory node that invoked the procedure.

The new lookup operation still searches for only one contact address. There is no *min* or *max* parameter to specify a desired number of contact addresses, as indicated in the location service interface in Chapter 2. Extending the *lookupAddress* procedure with this functionality is, however, straightforward. To easy our discussion, we therefore focus on searching for only a single contact address.

The procedure uses five local variables. The contact record *cr* stores a copy of the contact record of the specified object. The node identifier *child* stores the identity of the contact field from which we are retrieving a contact address or forwarding pointer. The contact address *addr* stores the oldest contact address found in this node. The variable *storedAddress* stores the contact address and associated information retrieved from another node. The variable *childPtrs* is used to hold the identities of the contact fields that hold a forwarding pointer.

The *lookupAddress* procedure consists of at most five steps. If a step is successful, the steps following it do not need to be tried.

1. Retrieve a contact address from the local contact record.

2. Use a cached node reference from the local domain to retrieve a contact address.

3. Follow a forwarding pointer in the local contact record to retrieve a contact address.

4. Use a cached node reference from outside the local domain to retrieve a contact address.

5. Contact the parent node to retrieve a contact address.

The first, third, and fifth steps were also present in the simplified lookup operation of Chapter 2.

The procedure starts, in line 10 in Listing 4.10, by retrieving a copy of the local contact record of the specified object. If the object is known in the current domain, a contact record is present, and the procedure can perform step 1–3, in lines 12–34. Otherwise, the procedure can skip ahead to step 4–5, in lines 37–51 in Listing 4.11.

In step 1, in lines 12–17, the *lookupAddress* procedure tries to retrieve a contact address from the local contact record. It specifically tries to retrieve the oldest address in the contact record, in line 13. This choice is based on the heuristic that an old contact address is a stable address. A downside to using the oldest address is that it forces all lookup procedures at a node to use the same contact address. Another option is to pick a random contact address from the local contact record, as was done in the simplified lookup procedure in Chapter 2. If an address is present (line 14), it is retrieved from the record and returned to the caller, in lines 15–16. The *storedAddress* return value is made up of the contact address, age of the address, and the identifier of the current node. If an address was found the procedure is ready.

```
procedure lookupAddress(oh : ObjectHandle,
                                  subDomOnly : Boolean,
                                  caller : NodeID) return StoredAddress is
  cr : ContactRecord;
  child : NodeID;
  addr : Address;
  storedAddr : StoredAddress;
  childPtrs : set of NodeID;
begin
  cr := crDatabase(oh);
  if cr $\neq$ NIL then
    -- Step 1: Try to retrieve an address from the local node.
    child := oldest in cr with cr(child).addrSet $\neq \emptyset$;
    if child $\neq$ NIL then
      addr := oldest in cr(child).addrSet;
      return (addr, date of addr, thisNode);
    end if;

    -- Step 2: Try to retrieve an address using local references from the cache.
    storedAddr := check_cache(object, local);
    if storedAddr $\neq$ NIL then return storedAddr end if;

    -- Step 3: Try to retrieve an address by following forwarding pointers.
    childPtrs := {child $\in$ index of cr with cr(child).isPtr};
    while childPtrs $\neq \emptyset$ loop
      -- Pick the child which forwarding pointer has been stored the longest.
      child := oldest in cr with child $\in$ childPtrs;
      childPtrs := childPtrs $-$ {child};
      storedAddr := call lookupAddress(oh, subDomOnly, thisNode) at child;
      if storedAddr $\neq$ NIL then
        cache_insert(object, storedAddr);
        return storedAddr;
      end if;
    end loop;
  end if;
```

Listing 4.10: The new *lookupAddress* procedure, part 1.

```
(36)     if not subDomOnly then
(37)        −− Step 4: Try to retrieve an address using a remote reference from
(38)        −− the cache.
(39)        storedAddr := check_cache(object, remote);
(40)        if storedAddr $\neq$ NIL then  return storedAddr end if;
(41)
(42)        −− Step 5: Try to retrieve a contact address through the parent node.
(43)        −− So far nothing has been found. Forward the request to the parent
(44)        −− thus broadening the search region.
(45)        if caller $\neq$ parent then
(46)           storedAddr := call lookupAddress(oh, false, thisNode) at parent;
(47)           if storedAddr $\neq$ NIL then
(48)              cache_insert(object, storedAddr);
(49)              return storedAddr;
(50)           end if
(51)        end if
(52)     end if
(53)     return NIL;
(54)  end lookupAddress;
```

Listing 4.11: The new *lookupAddress* procedure, part 2.

In step 2, in lines 19–21, the procedure tries to retrieve a contact address from the local domain using the location cache. It uses, in line 20, the *check_cache* procedure, described above, to do the actual work. The *lookupAddress* procedure specifies, using the *strategy* parameter, that it wants to use a reference to a node from the local domain. If a contact address was found, the *storedAddress* return value can be returned to the caller, and the procedure is finished, in line 21.

In step 3, in lines 23–34, the *lookupAddress* procedure tries to retrieve a contact address from the local domain using the forwarding pointers found in the contact record. The procedure first creates *childPtrs*, the set of node identifiers identifying the contact fields with forwarding pointers, in line 24. The procedure then loops, in lines 25–34, over this set, trying the oldest forwarding pointers first. The notion here is that an old forwarding pointer indicates a stable contact address. The body of the loop consists of retrieving the oldest forwarding pointer, in line 27, and removing it from the set, in line 28. The procedure then performs an RPC invocation to obtain the contact address at the child specified by the forwarding pointer, in line 29. If a contact address was found (line 30), the associated directory node is put in the location cache, in line 31. The result is returned to the caller, in line 32, and the procedure is finished.

Step 4 and step 5 are executed only if the *subDomOnly* parameter is *false*. The parameter is set to *true* when a node reference from the location cache is followed. When the *subDomOnly* parameter is set to *true*, the lookup operation is contained within the domain of the directory node that received the lookup request. The lookup operation cannot leave the domain since neither the use of remote cache references nor the parent node is allowed. This way we guarantee the termination of the lookup operation. A client starts the lookup operation at a leaf node with the *subDomOnly* parameter set to *false*.

In step 4, in lines 37–40 in Listing 4.11, the procedure uses the location cache for a second time. This time it instructs, in line 39, the *check_cache* procedure to use node references to directory nodes outside the current domain to obtain a contact address. If a contact address was found, the *storedAddress* return value can be returned to the caller, and the procedure is finished, in line 40.

In step 5, in lines 42–51, the *lookupAddress* procedure tries to obtain a contact address by invoking the *lookupAddress* procedure at the parent node, in line 46. This can be done only if the procedure was not invoked from the parent node in the first place (line 45). If a contact address was found by the parent (line 47), a reference to the node that stores the address is inserted in the location cache, in line 48, and the procedure returns the contact address, in line 49.

If none of the steps were successful, the *NIL* value is returned to the caller, in line 53, to indicate failure.

4.5 Simulation Results

To evaluate the effectiveness of our location caching mechanism, we performed a simulation experiment. The goal of this experiment was to show that our method of storing contact addresses higher in the tree and using location caches to store references to nodes where contact addresses were found improved the performance of lookup operations. Furthermore, we expected this improvement to be larger than the improvement provided by ordinary caching schemes.

4.5.1 Methodology

Since each object is handled independently by the location service, we simulated only a single object. This mobile object had, however, different mobility domains during its lifetime, resulting in movement across larger and smaller distances at different times. To generate the mobility and lookup activity, we used the same object models as in the simulation experiment in Chapter 3. The mobility and lookup patterns are shown in Table 4.1. The mobility pattern was used to choose the mobility domains of the object. The lookup pattern was used to choose the location of clients initiating lookup operations.

The characteristics of the search tree were also based upon the simulation experiment in Chapter 3. The logical search tree had the same four levels: a root level, a subcontinent level, a country level, and a city level. To determine at which physical nodes contact records would be stored for a logical node, we used the location-aware load distribution scheme. To simplify our simulation, we chose a fixed home location for the object in Amsterdam. Using the cities from the simulation from Chapter 3 and the fixed home location, we determined the cities that would have physical nodes for the object. We

Table 4.1: The probability distributions of the three object models used in the simulation experiment. G refers to global domains, and L refers to local domains. Model LM-GL thus stands for mobility only in the local domain and lookup operations initiated in the global domain.

Model ↓	Mobility				Lookup			
Level →	3	2	1	0	3	2	1	0
GM-GL	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
LM-GL	0.50	0.25	0.15	0.10	0.25	0.25	0.25	0.25
LM-LL	0.50	0.25	0.15	0.10	0.50	0.25	0.15	0.10

needed to determine these physical nodes to compute the geographical distances between nodes in the search tree.

In the experiment we simulated and compared the results of three types situations: no caching, data caches, and location caches. While comparing the situations, we looked at three properties of the location service: the total load experienced by all physical nodes, the total number of messages sent by all operations, and the total geographical distance traveled by all operations. The total load was measured by the total number of executed lookup and update procedures at all the nodes in the location service. For simplicity we assumed that the load of executing a single update and lookup procedure was equal. The total message count measured the number of RPC requests and replies sent. For simplicity we ignored other messages sent, such as acknowledgments or status requests. The total distance is measured as the distance traveled by all RPC requests and replies. We expected that with an effective caching mechanisms these properties would all decrease.

The no caching situation was the base line situation where lookup operations used tree traversals that followed the basic search tree structure to find contact addresses. In the data-caching situation, we simulated caches that stored contact addresses, with a data cache placed at every node in the search tree. If the lookup operation found a contact address in a data cache and the address was still valid, the operation was finished, and it returned the address to the client. If the lookup found a contact address that was no longer valid, the operation received a communication penalty. This communication penalty consisted of a pair of messages sent and the distance they traveled. Even if these messages were actually sent by the client, they were still the result of invalid cache entries in the location service. If no contact address was found in the cache, the lookup operation simply continued its traversal to find a contact address.

In the location-caching situation, a lookup operation always received a communication penalty when it found a cache entry since the operation always needed to communicate to check for a contact address at the node referred to in the entry. If the node referred to by the cache entry stored a contact address of the object, the lookup operation was finished. If the node stored a forwarding pointer, the operation followed that pointer to retrieve the contact address and then the operation was finished. If the node was found to be empty, the lookup operation continued its tree traversal at the original node. The penalty for an invalid location-cache entry thus consisted of a pair of messages sent, the distance they traveled, and an increase of the load at the receiving node. In the location-caching situation contact addresses were also moved down by the move-down operation. This generated extra load and communication costs.

In both the data-caching and the location-caching situation, the lookup operation also removed a cache entry when it was found to be invalid. Since it was unclear whether we could find an additional mechanism to purge invalid entries from the cache without actually verifying the entry, for instance, using the Alex cache replacement policy mentioned earlier, we simulated for both the data and the location cache a simple and an ideal case. In the simple case no additional mechanism was used to purge invalid entries from the cache. The ideal case represented a system where an additional mechanism existed that purged entries from the cache the moment they became invalid. We therefore looked at five situations in total: no caching, data caching, location caching, ideal data caching, and ideal location caching.

Since the move-up and the move-down operation used timing information, we also needed to simulate time in our experiment. In the simulation we used an abstract time unit that represented a fixed amount of real time, for instance, a period of several hours. In every time unit, only a single operation took place. Since we used the same mobility-to-lookup ratio of 0.2, as in Chapter 3, the object would move on average once every five time units and be looked up four times for every time it moved. For every object model, we chose 20 times a mobility domain and used that mobility domain during 1,000 time units, resulting in a simulation of 20,000 time units (and thus operations) per object model.

For the location-caching situation, we also needed to simulate the moving up and down of contact addresses. To do so, we needed to choose values for the mobility threshold, stability threshold, and aging factor. To allow the location cache to work efficiently, we chose values that would result in the direct placement of a contact address at the stable address location. Since we expected movement within a mobility domain every five time units, we set the mobility threshold slightly higher, at eight time units. To ensure a stable contact address would start moving downward straightaway, we chose the stability threshold slightly higher than the mobility threshold, at ten time units. We disabled the aging factor ($\alpha = 1$) since its only purpose was to avoid (over)reaction to incidental movements, that is, movement not part of the regular mobility pattern.

To get a fair comparison between the caching schemes, we separated our simulation program in a trace-generation tool and a trace-execution tool. That way the location service reacted in all three caching situation to the same client and object activity. Since there was one event per time unit, the trace-generation tool simply generated 20,000 events. The tool started with adding an insert event that inserted a contact address of the object in Amsterdam. For every event afterward the generation tool chose at random whether the event would be a movement or a lookup. If the event was object movement, the tool chose a new location using the current mobility domain. It then added a move entry to the trace file. If the event was a lookup, the tool chose a client location using the lookup pattern, and added a lookup entry to the trace file. Every 1,000 time units, the generation tool chose a new mobility domain using the probability distribution associated with the mobility pattern. The trace-generation tool ended with adding a final delete event to clean up the contact address of the object.

The trace-execution tool actually simulated the location service using a particular caching scheme. It read the trace file, one event at a time, and performed the activity described in the event. A move event meant inserting a new contact address and deleting the old contact address. While performing these update operations, the tool increased the total load, message count, and distance traveled. A lookup event meant performing a lookup operation from the location given by the event. While performing the lookup operation, the tool increased the total load, message count, and distance traveled. After handling an event from the trace file, the trace-execution tool checked whether a move-down operation needed to performed.

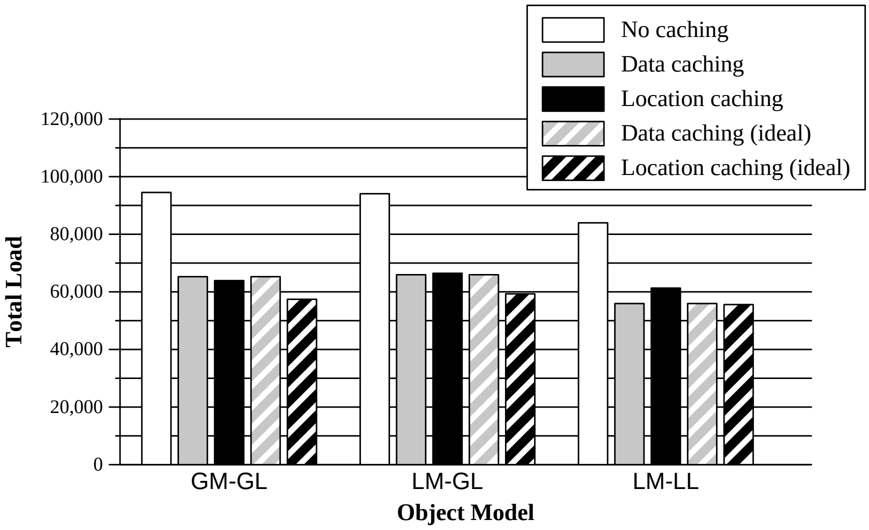
4.5.2 Results

Figure 4.7 shows the total load in the location service using the five caching schemes and three object models. The most obvious result is that both the data-caching scheme and location-caching scheme result in a significantly decreased load. In all object models the load is decreased by on average 30%. The fact that the improvement for the data cache is similar to the location cache is unexpected. It can, however, be explained by the fact that, in contrast to the location cache, invalid cache entries in the data cache do not result in a load increase. That cost is simply not visible in this figure since it is part of the communication between the user and the object which is not simulated.

The striped bars depict the load of using the ideal data and location cache. The result of the ideal data-caching scheme provides no improvement over the simple scheme. This is also caused by the fact that invalid data cache entries do not increase the load in the location service. The result of the ideal location cache is slightly better than the result of the simple data cache. Invalid cache entries in the simple location-caching scheme apparently do result in a slight decrease in performance.

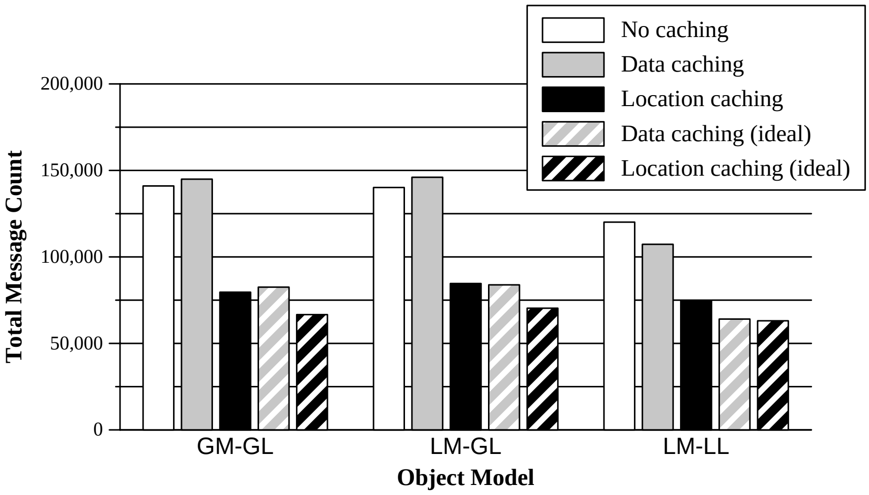
Figure 4.8 shows the total number of RPC messages sent. This figure is more revealing than the previous. It shows that the number of messages sent when using the data-caching scheme is similar to using no caching at all. A possible improvement of the number of messages is apparently completely negated by the number of messages sent in the case of invalid entries. The location-caching scheme shows in this case an improvement for all models of on average 40%.

The large impact of invalid entries in the simple data-caching scheme can be seen by comparing it with the ideal data-caching scheme. While the simple data-caching scheme results in a number of messages similar to the situation without a cache, the ideal data-caching scheme results in a drop in the number of message to at least 60%; a result comparable to the simple location-caching scheme. The ideal location-caching scheme, however, also provides an improvement over the simple location-caching scheme, although the improvement is less significant, on average an improvement of only 10%.



Model	GM-GL		LM-GL		LM-LL	
	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
No caching	94,513	100%	94,076	100%	83,960	100%
Data caching	65,266	69%	65,938	70%	55,943	67%
Location caching	63,904	67%	66,453	71%	61,316	73%
D. caching (ideal)	65,266	69%	65,938	70%	55,943	67%
L. caching (ideal)	57,421	61%	59,307	63%	55,588	66%

Figure 4.7: Total load generated by update and lookup operations using three types of workload and different caching schemes.



Model	GM-GL		LM-GL		LM-LL	
	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
No caching	141,020	100%	140,124	100%	120,104	100%
Data caching	144,950	103%	146,046	104%	107,292	89%
Location caching	79,609	56%	84,640	60%	74,528	62%
D. caching (ideal)	82,526	59%	83,848	60%	64,070	53%
L. caching (ideal)	66,643	47%	70,348	50%	63,072	53%

Figure 4.8: Total message count by update and lookup operations using three types of workload and different caching schemes.

Figure 4.9 is also revealing. It shows the distance traveled by update and lookup operations. It shows that the distance traveled by operations using the location cache is similar to the distance when using no cache. This is to be expected because with our location-aware load-distribution scheme going up the search tree and down again is roughly similar to going directly to the referenced node. In contrast, the data-caching scheme results in a distance that is on average 60% larger than the distance traveled without caching.

Comparing the ideal data-caching scheme with the simple data-caching scheme makes the communication penalty for invalid entries clearly visible again. What is more surprising is that the ideal data cache is as good or better (for the LM-LL model) than the ideal location cache. This can, however, be explained by the fact that the overhead for the location-caching scheme is higher. In the location-caching scheme, lookup operations always have to retrieve the contact addresses at the indicated node making it more expensive.

4.5.3 Conclusion

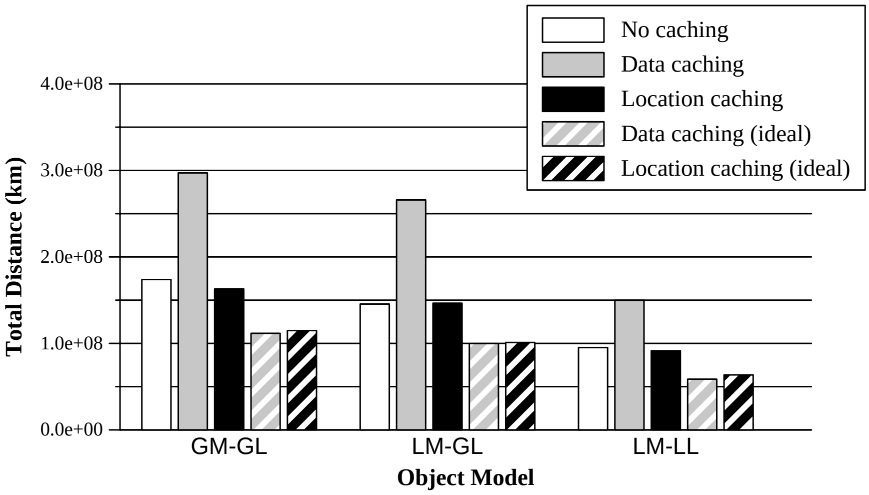
By introducing the location-caching scheme in the location service we can significantly improve its performance. Furthermore, this improvement is far more than that of a data-caching scheme. Our simulation clearly shows that a data cache is hampered by its invalid cache entries. Since both schemes benefit equally well from a scheme to remove invalid cache entries, it would seem likely that the location-caching scheme will continue to perform better. Only in the ideal case will the data-caching and location-caching schemes provide similar performance. It should be noted that these results are all based on an assumed workload. Our location-caching scheme still has to prove itself with real-life workloads.

4.6 Distinguishing Replicas

In this section we discuss an issue with the lookup operation that is independent of location caching and object mobility described so far.

Even though our objects can consists of multiple replicas, we have so far assumed that all replicas of an object are functionally equivalent and differ only in the location at which they reside. In a distributed system with replicated objects, we expect, however, that different replicas of an object play different roles. For instance, if an object uses a primary/backup replication strategy, it will have one primary replica and several backup replicas. When performing normal operations, a client can use any of the replicas, but if the client wants to add a new backup replica to the object, the client needs to know which of the replicas of the object is the primary replica. Another difference between replicas can be that one kind of replica supports only read operations while another kind supports both read and write operations on the object.

In the current design of the location service we cannot efficiently support looking up replicas that play specific roles. The only way to distinguish the primary replica,



Model	GM-GL		LM-GL		LM-LL	
	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
No caching	1.7e8	100%	1.5e8	100%	9.5e7	100%
Data caching	3.0e8	171%	2.7e8	182%	1.5e8	158%
Location caching	1.6e8	93%	1.5e8	101%	9.2e7	97%
D. caching (ideal)	1.1e8	65%	9.9e7	66%	5.8e7	61%
L. caching (ideal)	1.1e8	65%	1.0e8	67%	6.3e7	66%

Figure 4.9: Total geographical distances traveled by update and lookup operations using three types of workload and different caching schemes.

in the example above, is to associate the contact address of the primary replica with a second object handle. If a client is interested in only the primary replica, he will use this new object handle; for normal operations the old object handle is used. The solution is attractive since we can look for specific replicas without adding functionality to the location service.

There are, however, also serious problems with associating multiple object handles with a single object. These problems become visible when replicas can play multiple roles at the same time. The first problem is that the client needs to maintain multiple object handles to refer to an object since a single object handle is no longer enough to find all types of contact addresses. This problem also occurs in naming services that store the mapping from object name to object handle. An object name needs to be associated with all the object handles of an object. The number of object handles associated with an object name becomes particularly bothersome if the replicas of an object can play an increasing number of roles. Using multiple object handles also requires the client to do multiple lookup operations when he wants to lookup a replica without being interested in one specific role. The client also needs to determine which reply of the multiple lookup operations retrieved the nearest contact address. Introducing an object handle for every possible subset of roles the replicas can play is not attractive either since this choice would make the set of object handles associated with an object grow exponentially.

A generic extension to the location service to support different replica roles is to associate (attribute, value)-pairs with a contact address. Such pairs would be stored in the contact record together with the contact address. The client could add a predicate to the lookup operation specifying the attribute values of the contact address. The problem with this extension is that it violates locality. The location service needs to follow every path of forwarding pointers to the directory node that stores the contact address before it can decide whether the address will actually match the predicate.

It is, in principle, possible to add (attribute, value)-pairs and still support locality. To do this, we also need to associate the (attribute, value)-pairs of a contact address with the path of forwarding pointers pointing to the address. That way the lookup operation can already match the predicate with forwarding pointers higher up in the tree, and avoid traversing down the tree to the directory node that stores the contact address. We decided, however, that this generic solution would be too heavyweight for the location service due to its costly comparisons of attributes and the increased complexity of the lookup operation.

We therefore simplify the idea of generic (attribute, value)-pairs, and instead associate a **property map** with a contact address. A property map is a bit string where the bits in the bit string represent boolean characteristics of the object. The semantics of the properties are object specific, the value `1` simply means the replica has the property and the value `0` means the replica does not have the property. Since the semantics are object specific, different objects can assign different properties to the boolean values in the property map. When a client inserts a contact address, the client assigns a property map to the address. During the insert operation, the property map is associated with the contact address and every forwarding pointer on the path of forwarding pointers pointing to the

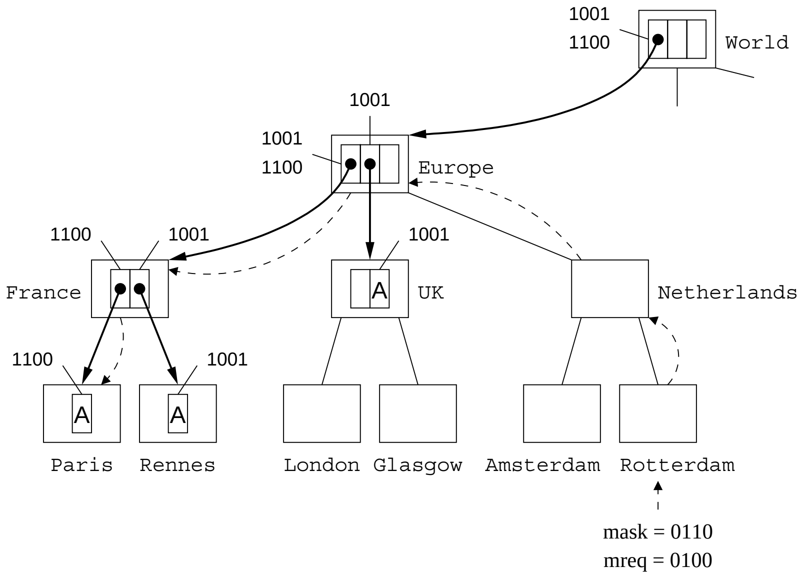


Figure 4.10: Property maps and a lookup operation in the search tree.

contact address.

For every lookup operation, the client specifies which bits of the property map he is interested in using a bit mask. The client also specifies the value each bit should have. The lookup operation returns a contact address (or follows a forwarding pointer) if the property map *m* associated with the contact address matches the request (*mreq*) in the following way.

$$(\textit{mask}\ \text{AND}\ m) = (\textit{mask}\ \text{AND}\ \textit{mreq})$$

Figure 4.10 shows an example of a search tree with property maps. The object in the example has three replicas. The property maps associated with the replicas consist of four bits. The contact addresses in the Rennes and Glasgow leaf domain have the property map `1001`, and the contact address in Paris has property map `1100`. At the European level, the contact record of this object has two forwarding pointers. The forwarding pointer pointing to the French domain has two property maps since the two contact addresses have different property maps. The forwarding pointer pointing to the UK domain has only one property map.

Consider the lookup operation in the example. It starts in the Rotterdam domain. The client is interested in a replica that has the second property and does not have third. The client is not interested whether a replica has (or does not have) the first and last property. To indicate that the client is interested only in the value of the second and third bit, the mask has a value of `0110`. The client requests (*mreq*) has the value `0100`. The lookup operation propagates up the tree until the `Europe` node where it finds the first contact record. The contact record contains two forwarding pointers, and the property maps of both are matched against the request of the client. The lookup operation follows the French forwarding pointers since it alone has a property map that matches against the client's request. At the `France` node only the forwarding pointer to the `Paris` node matches, and the lookup operation follows the forwarding pointer to the `Paris` node.

The ability to distinguish replicas during a lookup operation comes at the price of increased complexity and increased resource use of update operations. Since the property map of every contact address needs to be reflected in the property maps of forwarding pointers at higher levels in the search tree, every change to a property map at a low level needs to be propagated up the tree, potentially all the way up to the root node. Update operations therefore need to visit higher level nodes more frequently to make these changes. Previously, the upward traversal of an insert or delete operation was ready when a node was visited that stored a forwarding pointer, but with the new functionality, this forwarding pointer also needs to have the correct property map, otherwise the upward traversal needs to continue.

Chapter 5

Availability and Fault Tolerance

This chapter deals with the problem of ensuring that the location service is highly available. This availability depends heavily upon the way the location service deals with faults since faults can cause inconsistencies in the distributed state of the location service that, in turn, can result in the location service behaving incorrectly. The focus in this chapter is thus on making the location service fault tolerant, that is, ensuring it can maintain a consistent distributed state in the presence of faults. To keep the implementation of the lookup and update procedures simple, we want to shield the procedures from availability and fault tolerance issues as much as possible.

5.1 Failures

The location service plays a pivotal role in the Globe distributed system since it is involved in all communication. Since there are no other services in Globe that provide contact addresses, a client is (de facto) required to use the location service when he wants to communicate with an object. If the location service is unable to find a contact address, the client cannot bind to the object and communication cannot be initiated. When the location service fails to provide its service, the Globe distributed system breaks down since binding to objects is central to Globe. The functionality provided by the location service therefore needs to be highly available and highly reliable, that is, a client should be able to use the location service immediately with a high probability of success [Kopetz and Verissimo, 1993].

Faults within the distributed system, such as network partitions and crashing nodes, can lead to errors in the location service, such as the loss or corruption of location information, threatening its consistency and thereby its availability. Faults can occur in the location service itself or in the resources it uses, for example, the underlying network. Since the location service is a system consisting of many physical nodes and using a large number of resources, we can expect faults to occur frequently. While faults may lead to the failing of individual components of the location service, faults are not allowed to lead to a failure of the location service as a whole. Furthermore, the errors caused by faults should be repaired automatically while the service is in operation. Unfortunately, the process of masking faults in the location service can lead to a degradation in performance.

To structure our discussion, we make a distinction between *external* and *internal* failures with respect to a particular physical node. An external failure is the failing of a component of the location service that lies outside the physical node, for instance, a network connection or some other physical node. Even though the physical node is only indirectly affected by the failure, it still needs to deal with the consequences. The node can, however, rely on the correctness of its own state. An internal failure is the failing of (parts of) the physical node itself, for example, its software crashes or its hardware fails. In this case the physical node has to deal with the crash itself and the resulting inconsistency of its own state.

In the rest of this chapter, we first examine the problem of external failures, and describe how we deal with them. We then examine the problem of internal failures. We finish this chapter with a description of how we have implemented fault tolerance.

5.2 External Failures

This section describes how a physical node deals with the effects of external failures, such as network partitions and other physical nodes crashing. We also explain the important role concurrency plays for high availability when dealing with external failures. We finish this section with the description of a data structure we developed to ease the implementation of concurrent update and lookup procedures.

5.2.1 Problem Analysis

External failures are visible at a physical node as the failure of another node to respond to an RPC request, that is, the physical node simply keeps on waiting for an RPC reply. Since we assume a fail-silent failure model [Laprie, 1995], a physical node will not send or receive any incorrect request or reply. The failure to get the reply can originate at two places: in the network during the transmission of the RPC request or reply or at the other physical node during execution of the requested procedure.

The simplest cause of the failure to receive a reply is intermittently dropped network packets. This can happen during the transmission of RPC requests or replies. Fortunately, two cooperating physical nodes can easily deal with dropped packets using a combination of a time-out mechanism, packet retransmission, and sequence numbers. Furthermore, this kind of failure results only in a short delay in receiving the RPC reply, but has no significant impact on the processing of operations in the location service as a whole. We therefore ignore such failures in the rest of this chapter.

In contrast, network partitions and crashed nodes are more serious problems. Both types of problems are visible at a physical node as the absence of replies to all outstanding requests over a long period of time. This absence, however, should not prevent the location service from continuing to perform its task, and this absence should thus be dealt with. The location service handles the absence of replies for lookup requests differently from update requests because lookup and update requests have different requirements.

The location service needs to handle lookup operations quickly since there is always a client waiting for the answer. A physical node should therefore not wait indefinitely for a lookup RPC to finish, but instead place a timeout on the waiting period and abort the lookup RPC when it takes too long to complete. Aborting the lookup RPC is not a problem since the lookup operation can always continue by searching for contact addresses somewhere else in the search tree, for instance by following a different forwarding pointer or by going to a higher-level node. The downside of using a timeout mechanism in the lookup operation is that when the network is slow, we risk the location service not finding a contact address that it has actually stored. We prefer a fast lookup operation, however, over the certainty that the lookup operation will always find a contact address.

In contrast to lookup operations, update operations have less severe timing requirements since update operations do not have a return value that is to be used after the operation ends. The receipt of an RPC reply from an update operation simply tells the client that the operation has finished; unlike the lookup operation, the RPC reply does not contain a return value for the client to use. To guarantee consistency in the location service, however, we do require that all update RPCs are allowed to finish. Since we therefore cannot abort a long running RPC, a physical node potentially needs to wait a long time for before it can continue executing an update procedure. The only thing left to do when dealing with an unresponsive node is thus to save all outstanding update requests until the unresponsive node becomes responsive again and RPC replies can be received. If the (formerly) unresponsive node was unresponsive because the network was partitioned, it can simply filter out any duplicate request. If the unresponsive node had actually crashed, however, a recovery process is started, which is fully described in Section 5.3.

Unfortunately, the physical node sending the update request cannot simply wait for the update replies to come in and do nothing in the meantime; clients are sending new lookup and update requests continuously and are waiting for their results. To ensure the availability of the location service, physical nodes therefore have to handle incoming requests concurrently. This is especially true since these new requests do not necessarily require communication with the unresponsive node. Recall that different object handles can be handled by different physical nodes of a logical node.

A different problem is that a physical node cannot continue saving update requests for an unresponsive node indefinitely. The physical node will at a certain time run out of resources (first main memory and later disk storage) to store the RPC requests. A mechanism is thus needed to control the flow of RPC requests between physical nodes. This mechanism needs to ensure that a physical node cannot overflow another physical node that is saving RPC requests. A straightforward way to accomplish this is to place a limit on the number of concurrently running procedures per physical node. Once this limit is reached, the physical node will only accept RPC replies for update operations.

5.2.2 Concurrency

As stated in previous chapters, it is easy to handle RPC requests for different objects concurrently. Since there is no dependency between object handles, there are no consistency requirements to take care of. The concurrent handling of lookup requests for the same object is also not a problem since no changes are made. The real problem is how to deal with groups of update requests or groups of lookup and update request for the same object in a concurrent fashion. Note that these requests need not all come from the same child node.

By handling multiple update requests for the same object concurrently, multiple executing procedures use and possibly change the same contact record. Since this leads to race conditions and may result in inconsistent contact records, we give an executing procedure exclusive access to the contact record it uses (i.e., we lock the contact record). Unfortunately, as noted above, the execution of an update procedure might take a long time when other physical nodes are unresponsive. Since we do not want the contact record to be inaccessible for other requests during this period, we make the following exception with respect to mutual exclusion.

The exception is that during an RPC (i.e., when the node is waiting for a reply), we unlock the contact record, and a new procedure is allowed to start using the contact record or an old procedure that has received its RPC reply is allowed to continue using the record. In other words, the procedures and contact record together form a monitor [Hoare, 1974]. To ensure a correct end result of the concurrently executing procedures, we introduce two scheduling rules. First, if two update requests for the same object handle come from the same child node, the parent node must start the execution of the requested procedures in the same order that the RPC requests were sent. Second, if two concurrently running update procedures use the same contact record and are waiting for their respective RPC to finish, the procedure that sent its RPC request first must continue executing first as well, possibly delaying the continuation of the execution of the other procedure if its RPC reply was received first. The concurrent procedures are thus executed in a pipeline fashion.

For example, consider two update requests for the same object sent to a physical node by the same child node. The order of the requests is an insert request followed by a delete request. Following the first scheduling rule, the node starts with the execution of the insert procedure. The delete procedure starts its execution only when the insert procedure performs an RPC or has finished its execution. If, at a later time, both procedures are waiting concurrently for their respective RPC to finish, the insert procedure continues first after its RPC, that is, before the delete procedure continues, as stated by the second scheduling rule.

If two concurrent update requests come from different child nodes, the order in which the procedures are started is undefined. This does not affect the consistency of the contact record since the modifications use different contact fields and are thus unrelated. The order can, however, influence the final state of the search tree, as shown in Figure 5.1. In this example the height at which a new contact address is stored depends upon the order in which two concurrent update operations, a delete operation initiated in London and an

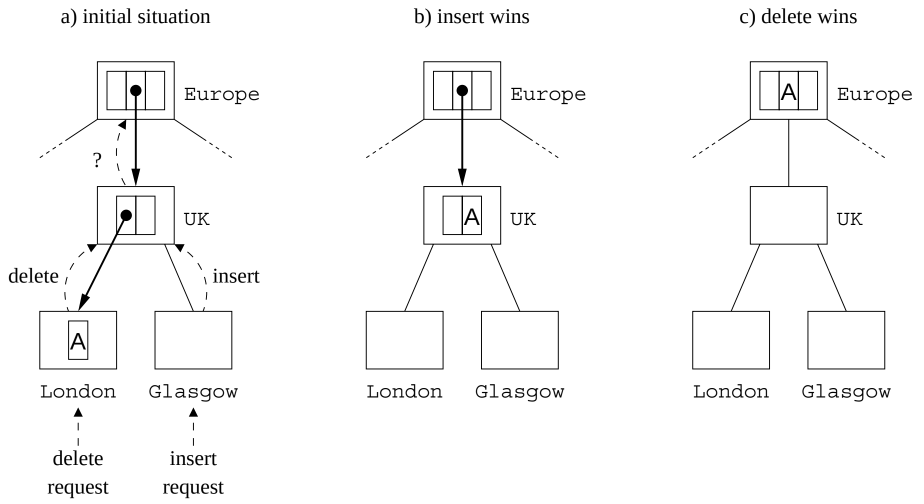


Figure 5.1: A race condition between a delete and an insert operation resulting in different search trees.

insert operation initiated in Glasgow, are executed. Recall from Chapter 4 that the height at which an address is stored is determined by a distributed decision making process, and that this process is based on the evaluations of individual nodes of whether they are a good node to store the contact address. In the figure we assume that, besides the `Glasgow` leaf node, the `UK` node and the `Europe` node also consider themselves a good node to store the contact address.

Figure 5.1(a) shows the initial situation with both leaf nodes sending an update request to the `UK` node. Figure 5.1(b) shows the end result if the `UK` node executes the insert procedure first. In this case, the `UK` node stores the new contact address since it is the highest available node that wants to store the contact address; then, the old address in the `London` node and its forwarding pointer in the `UK` node are deleted. Figure 5.1(c) shows the end result if the `UK` node executes the delete procedure first. In this case, the contact address in the `London` node and its forwarding pointers in the `UK` and `Europe` nodes are deleted. Afterward, the insert operation will thus propagate to the `Europe` node, which will subsequently store the contact address itself since it is the highest available node. It is important to note that both end results are valid.

Apart from update requests, a physical node also receives lookup requests. These lookup requests can be handled concurrently with update requests by simply abiding by the exclusive access rule. Since the lookup procedures do not make any changes, no ordering between a lookup and an update procedure is needed. We would, however, like the changes made to contact records by update procedures to become visible to lookup procedures as soon as possible. For instance, a contact address that is currently being inserted in the location service can be returned to a client immediately by a lookup procedure since it is for the client irrelevant whether the contact address will ultimately be stored at the current node or somewhere higher up the tree.

5.2.3 View Series

With the introduction of the concurrent handling of update requests, a contact record can have many tentative changes made to it concurrently. Keeping track of all these changes can easily increase the complexity of the lookup and update procedures. This is in conflict with our goal to keep our lookup and update procedures as simple as possible. We therefore introduce a new data structure, called a **view series**, to maintain the state of the concurrent changes to a contact record and keep the lookup and update procedures simple.

A view series is an association between a contact record and a queue of tentative changes to that contact record. Update procedures append these tentative changes to and remove them from a view series. View series maintain a strict FIFO ordering among the tentative changes: New tentative changes are appended at the tail of the queue and only the change at the head of the queue can be removed. When a tentative change is removed from the queue in the view series, the change is either made permanent by applying it to the associated contact record or it is discarded. Hence, the change is called *tentative* while it is in the queue, and *authoritative* when it is applied (permanently) to the contact record.

The view series allows update and lookup operations to compute the **current view** of the view series. The current view on the view series is the state of the contact record when all tentative changes in the queue are applied to it. The terms tentative and authoritative also apply to the current view. When a view series has changes in its queue, its current view is called tentative; otherwise, the current view is called authoritative.

Figure 5.2 shows the state of an example view series changing over time. The figure shows three columns. The left column shows the current state of the contact record that is associated with the view series. This state is stored on disk. The contact record consists of three contact fields. The middle column of the figure shows the tentative changes appended to the view series. Changes are appended to the right and removed from the left. These changes are stored in main memory. The right column shows the current view of the view series.

Figure 5.2(a) shows the initial situation. The contact record has a forwarding pointer in the left contact field, contact address A_1 in the middle contact field, and an empty right contact field. The view series has one tentative change appended to it. This change is the insertion of address A_2 in the middle contact field. The current view of the view series shows a contact record with a forwarding pointer in the left contact field, contact addresses A_1 and A_2 in the middle contact field, and an empty right contact field.

In Figure 5.2(b) a new tentative change is appended to the view series. The change is the deletion of the forwarding pointer in the left contact field. This does not change the contact record on disk, but does change the current view, as can be seen in the right column. In Figure 5.2(c) and Figure 5.2(d) the changes in the view series are applied to the contact record on disk. The changes are removed from the head of the sequence. Note that although the contact record on disk changes, the current contact record view remains

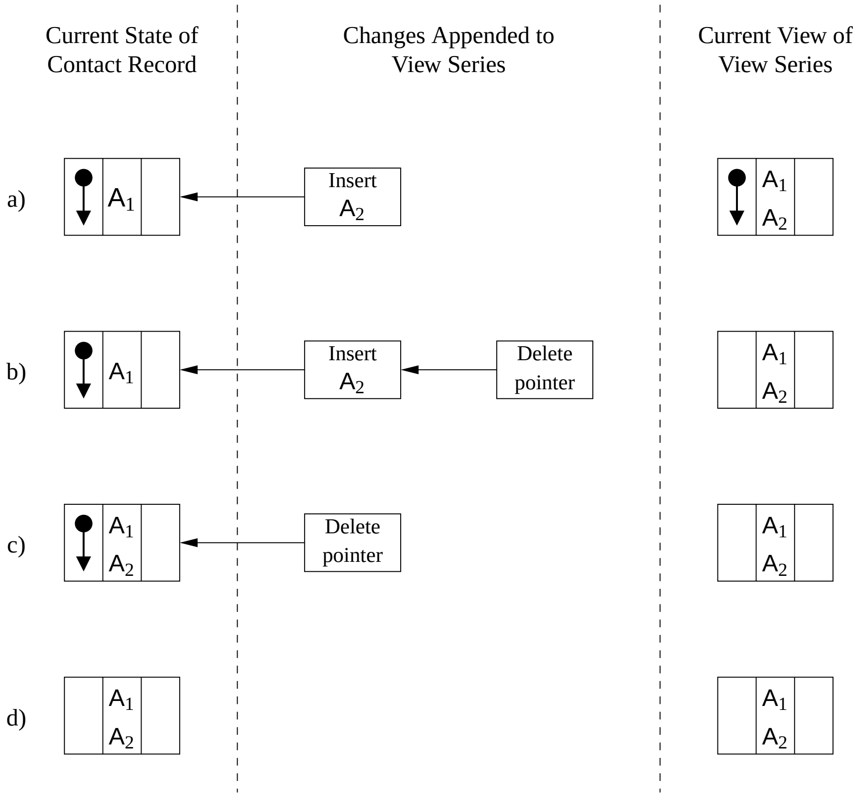

Current State of Contact Record
Changes Appended to View Series
Current View of View Series
a)
b)
c)
d)
A_1
A_2
Insert A_2
Delete pointer

Figure 5.2: Changes to a view series over time.

the same. In Figure 5.2(d) there are no changes appended, and the contact record on disk is the same as the current contact record view. In this case the current view of the view series is authoritative.

The tentative changes appended to the queue of the view series are the same changes normally made directly to a contact record. Thus, all changes made to a contact record, that is, adding or removing a contact address and setting or clearing a forwarding pointer, can also be appended to a view series. To simplify our procedures, we made the view series concept part of our Ada-like pseudocode. The following example shows a statement to append a new tentative change to a view series.

```
append view ⟨self(caller).addrSet :=
                    self(caller).addrSet + {addr}⟩ to tentativeCR;
```

In this example the variable *tentativeCR* is a view series. The tentative change appended to this view series adds the contact address *addr* to the contact record. To refer to the current value of the contact record, the view series uses the pseudo-variable **self**. This variable refers to the contact record at the time the change will be made permanent. The tentative change at the head of the queue (i.e., the least recently added change) is made permanent by applying it to the contact record. This is done using the following statement.

```
apply view to tentativeCR;
```

If the change at the head of the queue is unwanted, the change is removed from the queue of the view series. This is done using the following statement.

```
remove view from tentativeCR;
```

If the tentative change that is removed is, for example, the change append above, the contact address *addr* would no longer be visible in the view series. To obtain the current view of the view series, we use the following statement.

```
viewCR := view tentativeCR;
```

This statement computes a new contact record *viewCR* by making a copy of the contact record associated with the view series *tentativeCR*, and applying all appended changes to this copy. We use the *sizeOf* function to determine whether the current view of a view series is authoritative.

```
count := sizeOf( tentativeCR );
```

The *sizeOf* function returns the number of tentative changes appended to the view series. When the function returns zero, the view series is authoritative; otherwise, it is tentative.

Line 1 in Listing 5.1 shows the definition of the view series data type, called *ContactRecordView*. To keep track of all the view series currently in use in a physical node,

```
(1) type ContactRecordView : view series of ContactRecord;
(2) type CRViewSet : set (ObjectHandle) of ContactRecordView;
(3)
(4) tentativeCRSet : CRViewSet;
```

Listing 5.1: Data structure and global variable for the use of view series.

we define an indexed set of view series, in line 2. This type is used to define the global variable *tentativeCRSet*, in line 4. The lookup and update procedures use the procedure *getTentativeCR* to retrieve a view series from the *tentativeCRSet* variable.

To explain how view series are used in our update procedures, Listing 5.2 shows the new *insertAddress* procedure using view series. The structure of the procedure is very similar to the previous version, presented in Chapter 2. The procedure starts, in lines 9–10, by obtaining the view series of the contact record and computing the (initial) current view of the view series. In lines 13–14, the contact address is inserted by appending a tentative change to the view series.

As in the previous version, the procedure has to decide whether this node should obtain permission from its parent to store the contact address. The node should obtain this permission if the contact record was empty before it inserted the contact address. However, the node cannot be certain the contact record is nonempty if the current view (computed before the address was inserted) is only tentative. The procedure should therefore always obtain permission from the parent when the view series has appended changes.

The procedure uses the *sizeOf* function to determine whether there are appended changes. If the *sizeOf* function returns a value larger than one, concurrent update procedures have already appended other views to the view series, and we cannot be sure whether the contact record is actually nonempty. If the *sizeOf* function returns a value of exactly one, the current view of the contact record is authoritative, and we can be sure the contact record is nonempty. Note that the *sizeOf* function returns a value greater or equal to one since this procedure has appended a tentative change itself.

In line 18 the procedure decides whether it should store the contact address itself. If the contact record was already filled or the node wants to store the contact address, the procedure requests permission from the parent to store the address, in line 19; otherwise, the procedure requests the parent to insert the contact address itself, in line 20. If no approval is needed, the variable *approved* is set to *true*, in line 23. Depending on the variable *approved*, the procedure applies the tentative change to the contact record, in line 28, or it discards the change, in line 29. In both cases, the view that is appended in lines 13–14 is removed from the view series. The procedure returns *false* in line 33 to indicate the contact address was stored at this node or higher in the tree, as in the previous version.

The *insertAddress* procedure uses the procedure *getTentativeCR* to obtain the view series associated with an object handle. This procedure is depicted in Listing 5.3. The procedure starts, in lines 6–7, by checking whether the contact record and its view series are already in use. If so, the view series is stored in the global variable *tentativeCRSet*.

```
procedure insertAddress(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) returns Boolean is
  tentativeCR : ContactRecordView;          -- tentative contact record
  viewCR : ContactRecord;                   -- initial view of contact record
  approved : Boolean;                       -- parent approves address insert
begin
  -- Retrieve the contact record view series and the compute initial view.
  tentativeCR := getTentativeCR(oh);
  viewCR := view tentativeCR;

  -- Add the address (possibly temporarily) to the view series of the record.
  append view ⟨self(caller).addrSet :=
               self(caller).addrSet + {addr}⟩ to tentativeCR;

  -- Ask the parent for approval, if needed.
  if parent $\neq$ NIL and (isEmpty(viewCR) or sizeOf(tentativeCR) $> 1$) then
    if not isEmpty(viewCR) or doStoreHere(viewCR)
      then approved := call insertPointer(oh, addr, thisNode) at parent;
      else approved := call insertAddress(oh, addr, thisNode) at parent;
    end if;
  else
    approved := true;                       -- No approval needed.
  end if;

  -- Undo the local modification, if the parent does not approve.
  if approved
    then apply view to tentativeCR;
    else remove view from tentativeCR;
  end if;

  -- The contact address is stored here, or higher in the tree.
  return false;
end insertAddress;
```

Listing 5.2: The concurrent *insertAddress* procedure.

```
(1)  procedure getTentativeCR(oh : ObjectHandle)
(2)                                  returns ContactRecordView is
(3)    tentativeCR : ContactRecordView;                 -- tentative contact record
(4)    cr : ContactRecord;                                    -- contact record
(5)  begin
(6)    tentativeCR := tentativeCRSet(oh);
(7)    if tentativeCR = NIL then
(8)      -- The contact record and its view series are not yet used.
(9)      cr := crDatabase(oh);
(10)
(11)     -- Create and initialize a new contact record, if none exists.
(12)     if cr = NIL then
(13)       cr := new ContactRecord;
(14)       initializeCR(cr, children, thisNode);
(15)     end if;
(16)
(17)     -- Create and initialize the view series of the contact record.
(18)     tentativeCR := new ContactRecordView;
(19)     initializeCRView(tentativeCR, cr);
(20)
(21)     -- Make the view series available.
(22)     tentativeCRSet(oh) := tentativeCR;
(23)   end if;
(24)
(25)   return tentativeCR;
(26) end getTentativeCR;
```

Listing 5.3: The *getTentativeCR* procedure returns the view series of the contact record associated with the given object handle.

If no view series is available, a new view series is created and initialized, in lines 8–22. First, the procedure checks whether the object is known and has a contact record, in line 12. If the object is unknown, no contact record is stored in the local contact record database, and a new contact record is created and initialized, in lines 13–14. With the contact record available, a new view series is created and initialized, in lines 17–19. To make the view series available for new concurrent procedures, it is inserted in the global view series variable, in line 22. The view series is finally returned in line 25.

For the view series on the contact record to remain consistent, the procedure that appends a tentative change should also be the procedure that removes or applies that change, that is, the order in which procedures append changes should also be the order in which they remove or apply changes. This FIFO order is maintained using the second scheduling rule, described in the previous section. Since an update procedure always invokes an RPC when it encounters a view series with changes appended, the procedure will have to wait until the previous procedures have finished before it can continue itself. Its change is therefore always in the head position of the queue when it applies or removes the view.

Each insert or delete procedure consists of three phases:

1. Append a tentative change to the view series of the object. This is the change the procedure would like to make.

2. Ask the parent permission for the intended change. This allows the parent node to override the change or make the necessary changes itself to keep the search tree consistent.

3. Make the tentative change permanent or discard it by applying or removing the appended change. Applying the tentative change makes the change to the contact record authoritative.

Only the insert procedure decides whether to apply or remove its appended change; in contrast, the delete procedure always applies its change since the change (removing a contact address or clearing a forwarding pointer) always has to be done. After the third phase, the physical node sends the RPC reply that completes the procedure.

5.3 Internal Failures

This section describes how a physical node deals with the effects of crashing itself. In this section we assume a fail-stop system [Laprie, 1995] in which the physical node just stops working; no erroneous messages are sent and no erroneous data are written to disk before the node crashed. We also assume the use of atomic disk writes [Lampson and Sturgis, 1979] and node crashes that do not affect a disk. A node crash therefore results only in the loss of main memory.

The focus of this section is on repairing inconsistencies in the distributed state of the search tree. We start this section with a description of how a node crash can cause inconsistencies in the distributed state of the location service. We then propose a method of resolving these inconsistencies after a node crash. We finish this section with the reasons why our method is correct. In Section 5.4 we propose solutions to dealing with the loss of persistent state.

5.3.1 Problem Analysis

The execution of an update procedure in a leaf node usually involves making an RPC call to its parent node. The execution of the update procedure at the parent node, in turn, usually involves making an RPC to its parent as well, leading to a chain of cascaded invocations. An update operation therefore consists of a **chain of RPCs** from the leaf node upward, possibly to the root. This chain of RPCs is severed when one of the nodes on this chain crashes.

During a crash, a physical node loses all its in-core state, including information about the communication it is involved in at the time of the crash and the tentative changes

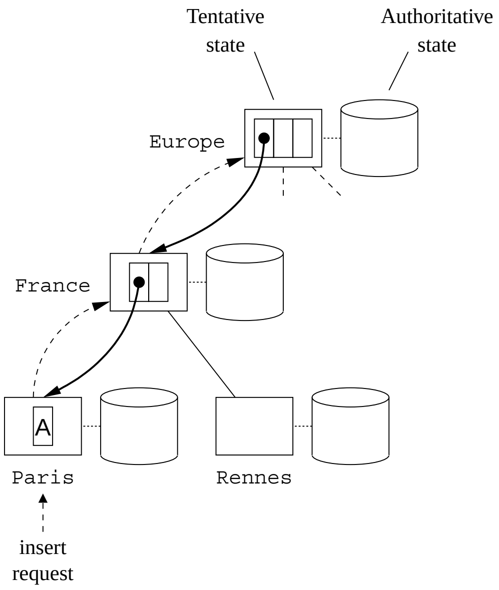


Figure 5.3: Crash during an insert operation, part 1. The insert operation insert the contact address and forwarding pointers tentatively.

stored in its view series. If the crashed node had sent update requests to its parent, the parent node will continue performing the requested update procedure since it is unaware of the crash. This executing procedure is called an orphan [Panzieri and Shrivastava, 1988]. During the execution of the orphaned procedure the parent will possibly modify its own contact record and send out an RPC request. The severed chain of RPCs thus results in nodes located higher in the tree than the crashed node performing their part of the update operation while the nodes located lower in the tree do not. This may result in inconsistencies in the tree, for instance, a path of forwarding pointers ending in an empty contact record.

Figures 5.3–5.5 show how a node crash during an insert operation can introduce an inconsistency in the search tree. In the figures we show both the current view of an object's view series and the state of its contact record on disk. In Figure 5.3 a client in the Parisian domain sends an insert request to the `Paris` leaf node. As a result, an insert operation is started that tentatively inserts a contact address at the `Paris` node and a forwarding pointer at the `France` and `Europe` nodes.

In Figure 5.4 the insert operation continues by making its tentative change at the `Europe` node authoritative, that is, saving the resulting contact record to disk. After the change is made authoritative, an RPC reply is sent back to the `France` node indicating that the RPC request has been handled at the `Europe` node. The operation continues at the `France` node upon receipt of this reply. The `France` node then makes its changes

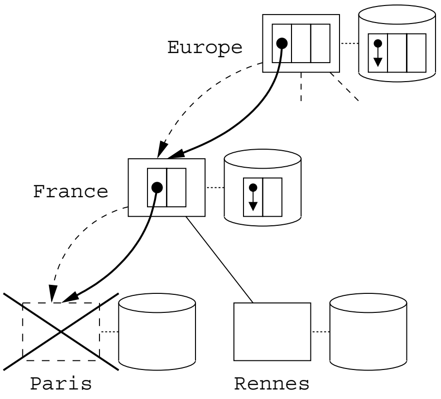


Figure 5.4: Crash during an insert operation, part 2. The insert operation makes tentative changes authoritative, but leaf nodes crashes.

authoritative, and sends its reply to the `Paris` leaf node. However, the `Paris` leaf node crashes before it can receive the RPC reply from the `France` node. The `Paris` node loses its tentative changes and forgets it was handling an insert request. Figure 5.5 shows the end result with the path of forwarding pointers in the `Europe` and `France` nodes pointing to the empty `Paris` leaf node.

The child nodes of a crashed node face a different problem caused by the node crash. To deal with intermittently dropped network packets, a receiving node (i.e., the parent) acknowledges receiving a packet. This tells the sending node (i.e., the child) that the packet has been received and that the RPC request or reply in the packet will be dealt with. However, if the receiving node crashes after sending the acknowledgment, the node loses all knowledge of having received the RPC request or reply while the sending nodes still assume that their packets will be dealt with. Especially problematic is the loss of an update RPC request from a child node since this results in the child node assuming its RPC request is being handled, and the child node will thus wait indefinitely for an RPC reply.

The purpose of the recovery mechanism is thus twofold:

1. The mechanism must correct the inconsistencies created by update operations broken off due to a node crash.
2. The mechanism must allow update operations to complete, even though nodes crash during their execution.

5.3.2 Crash Recovery

The crash recovery method must resolve the inconsistencies between the crashed node and its parent. However, the recovery method does not have to reconstruct the exact old state of the tree before the crash; any consistent state will do as long as it is a possible

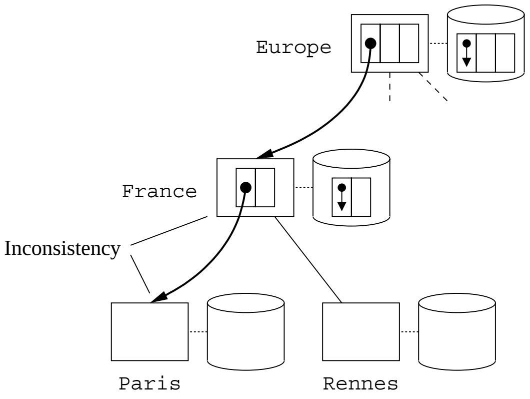


Figure 5.5: Crash during an insert operation, part 3. The `Paris` leaf node reverts back to its authoritative empty state and becomes inconsistent with its parent, the `France` node.

result of the authoritative state before the crash with the pending update operations applied to it. We are therefore interested only in ensuring that updates are not lost and that the consistency between a crashed node and its parent is restored. There can be more than one consistent state since the same set of update operations can result in a different state due to race conditions (recall the example of Figure 5.1).

The recovery method is based on the idea that since these inconsistencies were caused by the partial execution of outstanding update requests, they will be resolved when all outstanding update requests are restarted and fully handled. The method thus consists of the resending of the outstanding update requests by the children of a recovering node. Since these child nodes are still waiting for an RPC reply anyway, they can easily resend their requests when they notice the crashed node has restarted. The restarted node executes these resent requests, some possibly for a second time, and resolves the inconsistencies that might have arisen during the node crash. Note that we have to ensure that our update operations are idempotent. We discussed this and other requirements of our method in the next section.

The re-execution of the update procedures at the recovering node results in new update requests being sent to its parent node. The parent simply executes the requested procedures using the current version of its contact record, unaware that its child is actually recovering from a node crash and unaware that these requests were the result of the crash recovery method. The crashed node uses the reply sent by the parent node to become consistent with its parent.

Consider first the situation of re-executing an insert procedure at the recovering node. Three cases can be distinguished. First, if the contact field at the parent associated with the recovering child node already stores a contact address, the parent will store the new contact address as well, and tell the recovering node to discard its modification. Second,

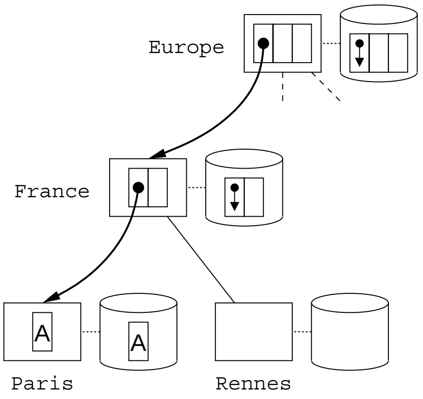


Figure 5.6: The occurrence of inconsistency, part 1. The initial situation

if the contact field already stores a forwarding pointer, the parent will tell the recovering node to store the contact address. Third, if the contact field is empty, the parent can choose what to do, and subsequently tell the recovering node what it should do. In all cases, the recovering node gets told by the parent what to do (i.e., keep or discard its modification) to become consistent. If an inconsistency between this node and its parent was caused by the insert operation during the node crash, it is now resolved. Note that the parent node does not have to be aware it is actually re-executing the insert procedure.

Now consider the situation where the recovering node re-executes a delete procedure. If the contact record at the recovering node does not become empty, the delete procedure is finished. If the contact record became empty or was already empty, the recovering node invokes a delete procedure at its parent. Two cases can be distinguished. Either the parent node already received a delete request for this operation from the recovering node or it did not. If the parent already received a request, the part of the delete operation above the crashed node is already executed. The procedure will try to delete the contact address or forwarding pointer and find it is already done. If the parent did not yet receive a delete request, it simply deletes the contact address and its path of forwarding pointers. Either way when the delete procedure is executed the path of forwarding pointers is deleted.

In principle we can use the same update procedures (i.e., the algorithms) to resolve inconsistencies as used in normal operations. Figures 5.6–5.11 show, however, that the inconsistency between a recovering node and its parent might pose a problem during the recovery of the node. Consider a part of a search tree with one contact address inserted at the `Paris` leaf node. This address is about to be deleted and a new contact address at the `Rennes` leaf node is about to be inserted, as shown in Figure 5.6.

If the recovery method consisted of simply resending the insert request the following might happen. In Figure 5.7 the delete request is received at the `Paris` node, and a delete operation is started that tentatively deletes the contact address and forwarding pointers. In Figure 5.8 the second half of the delete operation starts, and the delete operation makes

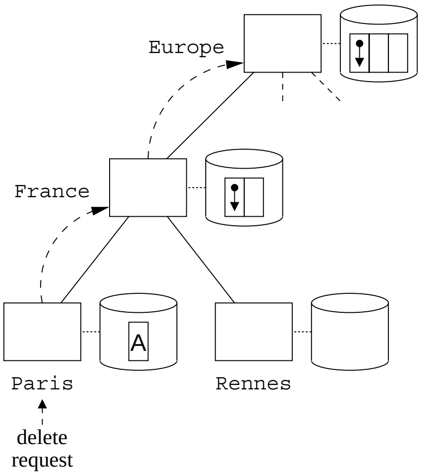


Figure 5.7: The occurrence of inconsistency, part 2. The delete operation tentatively removes the contact address and forwarding pointers.

its change at the `Europe` node authoritative and an RPC reply is sent. The `France` node crashes, however, before the reply is received.

The resulting situation is shown in Figure 5.9. The `France` node only has its authoritative state, and forgot it was deleting its forwarding pointer. The `Paris` leaf node is waiting for a reply on its delete request, and the `Rennes` leaf node is about to start its insert operation. Figure 5.10 shows the `Rennes` leaf node receiving its insert request, and starting an insert operation. This operation inserts tentatively a new contact address at the `Rennes` leaf node and a forwarding pointer at the `France` node. However, given that the forwarding pointer to the `Paris` leaf node is authoritatively stored at the `France` node, the insert operation assumes *incorrectly* that a forwarding pointer is also stored at the `Europe` node and immediately makes the new forwarding pointer also authoritative.

Figure 5.11 shows the resulting situation. The insert operation has finished by making the contact address authoritative. However, the delete operation, which is repeated as part of the crash recovery, deletes the forwarding pointer at the `France` node, and given the presence of the forwarding pointer to the `Rennes` node incorrectly assumes it is finished as well. The `France` node therefore sends an RPC reply to the `Paris` node allowing the delete operation to finish. Both the delete and insert operation have finished successfully, and assume that they have left the tree in a consistent state. The inconsistency between the `Europe` and `France` node in Figure 5.11 clearly shows that this is not the case. Update operations thus need to behave differently during the recovery since they cannot use the local contact record to determine what is stored at the parent node.

The problem with using (normal) update procedures to restore consistency is that they assume that the existing information in the search tree is consistent. However, if a node is

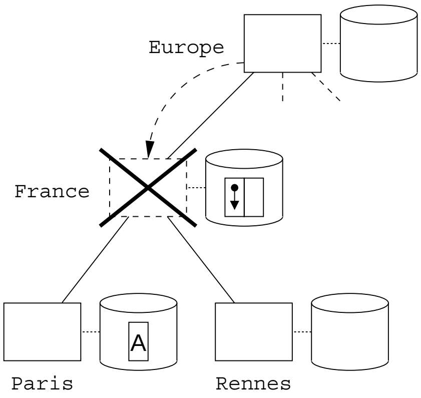


Figure 5.8: The occurrence of inconsistency, part 3. The delete operations makes the change in `Europe` node authoritatively, but the `France` node crashes before receiving the RPC reply from the `Europe` node.

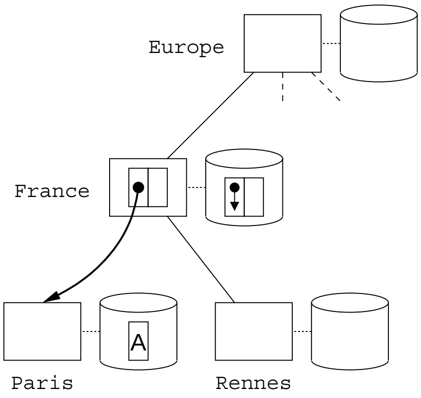


Figure 5.9: The occurrence of inconsistency, part 4. The `France` node reverts back to its previous authoritative state.

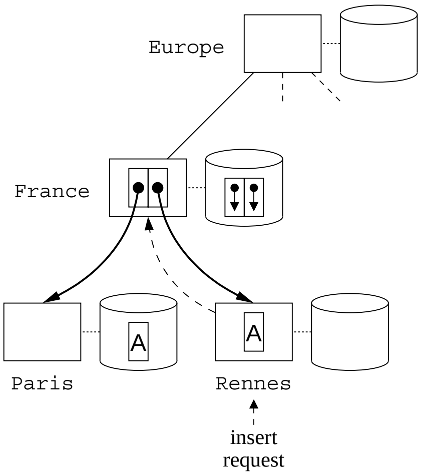


Figure 5.10: The occurrence of inconsistency, part 5. The insert operation insert a tentative contact address at the `Rennes` node and an authoritative forwarding pointer at the `France` node.

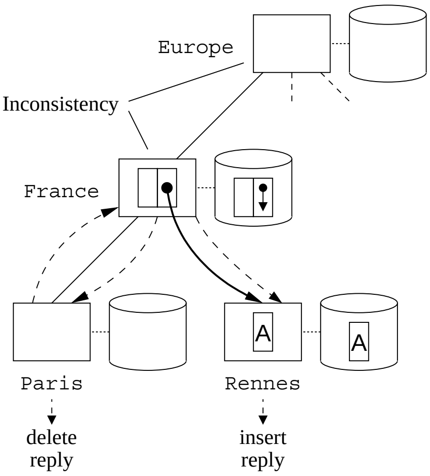


Figure 5.11: The occurrence of inconsistency, part 6. The delete operation removes authoritatively the forwarding pointer at the `France` node and the contact address at the `Paris` node. Both the delete and insert operation are completed, while the inconsistency remains between the `Europe` and `France` nodes.

recovering from a node crash, this assumption is obviously incorrect. More specifically, the problem is that an insert operation assumes that if it finds an authoritative nonempty contact record at a node, the parent node will have an authoritative forwarding pointer, and the parent node does not have to be contacted.

We call the period during which the crashed node is resolving inconsistencies the **recovery phase**. To avoid creating an inconsistent tree during the recovery phase, we modify the behavior of the insert operation during the recovery phase. The basic idea is that the insert procedure does not trust the local contact record and does not use it to draw conclusions about the contact record at the parent node. An insert procedure therefore *always* requests permission for its modification from its parent during the recovery phase. The parent node can then insert the address or a forwarding pointer if needed. Using the reply from the parent, the insert procedure can perform the correct local change, as described above.

The recovery phase is finished when all inconsistencies have been resolved, that is, all contact records at the recovering node are consistent with those at the parent. The problem is how to determine efficiently when this point is reached. Every inconsistency is caused by an update operation that was only partially completed, and for which the crashed node was awaiting an answer from its parent. Since that same inconsistency disappears when the node re-executes the update request, a node becomes consistent when it has executed all operations that were broken off by the node crash. Therefore, when each child of the crashed node has finished executing all *previously issued* update requests, the recovery phase is finished.

The recovery phase is implemented by distinguishing the RPC requests *resent* after a node crash as **recovery requests**. A special `end-recovery-requests` message signals the end of the stream of recovery requests and the start of regular (new) requests. If the recovering node has received `end-recovery-requests` messages from all its children, it knows that when all currently running update procedure are finished, its contact records are consistent with the records at its parent.

Unfortunately, in two cases there are no child nodes that can repeat their outstanding update requests. The first case is when the recovering node is a leaf node. Location service clients can, in principle, also resend their update requests, but this would make the location service dependent upon clients for its internal consistency. Since the organization maintaining the location service has no control over these clients this would increase the likelihood of errors, and using clients to resend update requests is thus undesirable. The second case is the take-over message, sent by the parent node as part of the move-down operation, described in Chapter 4. Since the parent retains no knowledge of having sent the take-over message (recall that it is only a message, not an RPC call), it cannot resend the request.

To deal with these two cases, every physical node has a persistent message log, for instance, on disk. This persistent log stores the update request messages received from clients by a leaf node and the take-over messages received by all nodes. These messages are kept in the log during the execution of the update procedure and are deleted once the execution has finished. When a physical node restarts, it starts handling the requests in the message log as if these were recovery requests sent by normal child nodes. When the old requests in the log are handled, the node continues with the requests it receives over the network. We use this persistent message log only in these two cases (instead for all update requests) since to guarantee consistency, the message logging to disk needs to be synchronous, making it an expensive operation.

5.3.3 Correctness

Our recovery method is similar to message-logging systems, such as sender-based message-logging [Johnson and Zwaenepoel, 1987], but does not need the mechanisms used in message-logging systems to repeat messages in strictly the same order as before the crash. Our recovery method can recover a consistent distributed state without such mechanisms because of three reasons:

1. A modification to a contact record is made permanent **atomically**.
2. Every modification to a contact record is **idempotent**.
3. Modifications to a contact record resulting from different child nodes are **commutative**.

The modified contact record is written to disk in an atomic fashion, that is, a node has either written a modified contact record completely before it crashes, or the node crashes before it started its write operation. A node never crashes during a write operation. Since we also assume that contact records already stored on disk are not corrupted by a node crash, we basically assume the use of stable storage techniques in the implementation of the contact record database. The recovery mechanism therefore does not have to worry about inconsistencies within a contact record.

The modifications to a contact record are idempotent since the contact fields that make up a contact record use (mathematical) sets to store contact addresses and boolean values to store forwarding pointers. As a consequence, inserting a contact address in or deleting it from a contact field for a second time does not change the contact field. Likewise, setting or clearing the forwarding pointer a second time does not change the contact field either. Being idempotent allows the modifications to be redone without adverse effect. The idempotency also applies to groups of operations since the update requests are retransmitted in the original order.

The modifications requested by different child nodes are commutative since different child nodes operate on different contact fields. Requests retransmitted by different child nodes therefore do not interfere. The final result, however, can be different than expected before the crash since the ordering of requests from two children can be different during the recovery. This is not a problem since the issue is not whether we restore the original state, but instead, that we end in a *consistent* state. Since update operations are commutative from different child nodes, any ordering of requests will result in a consistent state. Note, however, that the original ordering of requests *per child* does need to be maintained.

The recovery method requires that all update operations start at a lower node and work their way up the tree. That way the RPC reply signals that the nodes higher in the tree have performed their part of the update operation successfully. A physical node is therefore allowed to send its RPC reply only when it is sure its modification is safely written to disk. The write operation can thus be considered as making a checkpoint. The move-down operation is therefore also structured as an operation that starts at the child node and that sends an update request to its parent. The child node uses the information in the RPC reply from the parent to become consistent again.

An important aspect of the recovery method is that it is *transparent* to the update procedures executing at the parent and children of the recovering node. The update procedures at a child node are unaware of the RPC system sending the update requests to the crashed node for a second time. An update procedure requests an RPC only once, but when the RPC is completed, the procedure is guaranteed to have been performed at the parent, possibly more than once. The RPC system thus guarantees at-least-once semantics [Spector, 1982]. The parent node is unaware of receiving the same RPC request message multiple times; it deals with every request as a separate individual request. This does not lead to problems because the update procedures are idempotent.

It follows from this recovery transparency at child and parent that multiple simultaneous node crashes can be handled without extra effort. When two nodes on different paths from leaf to root crash, their recovery is unrelated. When two nodes on the same path from leaf to root crash, but these nodes are not directly connected, no problem arises as well. In both cases a crashed node has normal running child nodes (i.e., nodes that are not in the recovery phase themselves), which retransmit their update requests.

The question is what happens when two directly connected nodes (i.e., parent and child) crash simultaneously. In this case, the recovering parent will receive retransmissions from its grandchildren indirectly via its recovering child. During its recovery, the recovering child node will transmit its requests to the recovering parent, just as in normal recovery of the parent. The parent and child nodes are completely unaware of each other's recovery. When the recovering child has fully recovered itself, it will send its `end-recovery-requests` message to the parent node. Note that the recovery method can therefore also deal with the child node crashing during the recovery of the parent node. When the child is restarted, it will simply start resending the recovery requests (some possibly for the third time), while it is recovering itself.

Figure 5.12 shows two directly connected nodes, the `Europe` and `France` nodes, recovering from a node crash. Both nodes are assisted by the child nodes. The `France` node plays a special role since it is both recovering itself and sending recovery requests to the `Europe` node. The update procedures executing at the `France` node are, however, not aware that the `Europe` node is also recovering. The `France` node sends an `end-recovery-requests` message to the `Europe` node when it has received `end-recovery-requests` messages from the `Paris` and `Rennes` nodes and finished handling all update requests received before the `end-recovery-requests` messages.

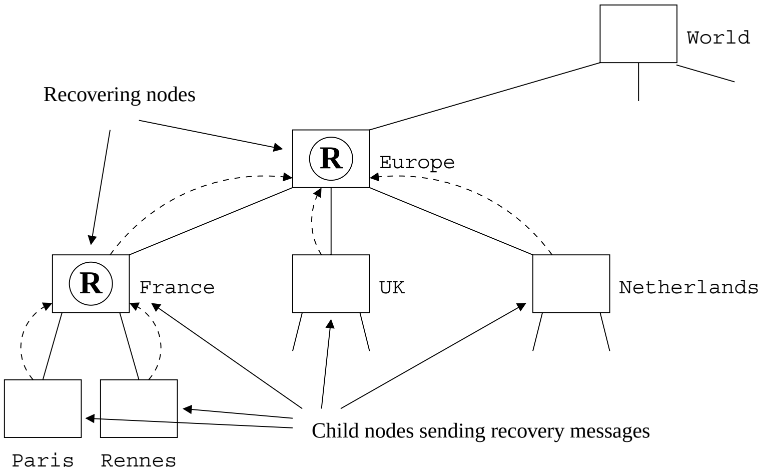


Figure 5.12: The `Europe` and `France` node recovering simultaneously. Recovery messages are sent two both nodes, and the `France` nodes helps the recovery of the `Europe` node while recovering itself.

5.4 Media Failures

In this section we describe some methods that can be used to deal with the loss of persistent data, that is, disk crashes. Since we have not investigated the problem of lost persistent state thoroughly, the discussion here is speculative and contains only some global ideas. To simplify our discussion, we assume in this section that the physical node has lost *all* its persistent data.

When a physical node loses its persistent data, all paths of forwarding pointers that used the node have become inconsistent. Two types of inconsistency problems occur. The first problem is inconsistencies between the crashed node and its children. This happens when a child node stores contact records and the crashed node no longer stores forwarding pointers to the child node. The second problem is inconsistencies between the crashed node and its parent. This happens when the parent has forwarding pointers pointing to the crashed node and the crashed node does not store any contact record. Furthermore, the contact addresses stored by the physical node are also lost.

The search tree can be made consistent again, but this might still include the loss of location information. The recovery process consists of two steps. The first step deals with the inconsistency problems between the crashed node and its children. The children go through their persistent contact record database and for every contact record tell the recovering node that it should store a forwarding pointer. This ensures that the contact addresses in the subtree of the recovering node can be found again.

The second step deals with the inconsistency problems between the crashed node and its parent. In this step the parent node goes through its persistent contact record database and for every forwarding pointer checks with the recovering node if a nonempty contact record exists. If no contact record can be found at the recovering node, the recovering node apparently used to store a contact address that has been lost due to the disk crash. Therefore, the path of forwarding pointers ending at the parent must be removed because the contact address is permanently lost. After these two steps, the search tree is consistent again.

Both steps in the recovery process are, unfortunately, heavyweight since they require significant communication between nodes and a traversal through the complete persistent databases at the parent and child nodes. Fortunately, we can make use of stable storage techniques to store data highly reliable in the presence of disk crashes. These techniques introduce reliability by storing data redundantly on multiple disks. These disks can be located at the same site, such as in RAID technology [Chen et al., 1994], or located at different sites, such as in the Petal system [Lee and Thekkath, 1996]. Using these techniques, the loss of a physical node is limited to the risk of catastrophic events, such as fires and floods.

Even though the recovery process can establish a consistent search tree, there remains the problem of dealing with the lost contact addresses stored at the recovering node. One way to deal with this problem is to reuse the contact address lease system, discussed in Chapter 2. Since a client is required to renew its lease regularly, the contact address can be reinserted when it appears missing during a lease renewal. The contact address is then lost only for the period between the disk crash and the renewal of the lease. A compromise has to be found for the duration of the lease: too long will result in a slow recovery of the contact address; too short will result in leaf nodes frequently handling renewal requests.

5.5 Implementation

This section describes how view series and crash recovery are integrated into the location service. It starts with an overview of the layers in the design of a physical node. The rest of this section presents the fault-tolerant versions of the update procedures. The lookup procedure can stay the same as presented in Chapter 4. It simply uses the current contact record view instead of the contact record itself.

5.5.1 Design

The design of a physical node with the crash recovery techniques implemented can be divided into five layers, as shown in Figure 5.13. The top layer contains the implementation of the lookup and update procedures. The second layer contains the implementation of the exclusive access and scheduling rules for executing procedures, as described in the beginning of this chapter. The third layer is the distribution layer dealing with node partitioning issues, as described in Chapter 3. The fourth layer is the RPC layer that is responsible for the high-level RPC communication. This layer is responsible for maintaining the ordering of RPC requests and replies and for resending RPC requests when a physical node restarts.

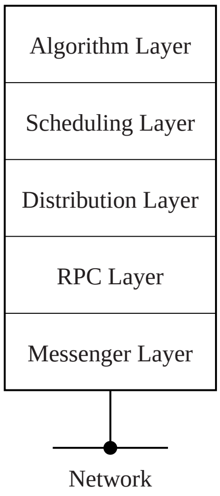


Figure 5.13: Layering in the design of a physical node

The layer implements the *inRecovery* function that indicates whether the node is in the recovery phase. It also maintains the persistent message log. The bottom layer is the messenger layer that is responsible for reliable low-level communication between nodes.

5.5.2 Insert Operation

Listing 5.4 and Listing 5.5 show versions of the *insertAddress* and *insertPointer* procedures that make use of view series and implement the crash-recovery extensions. There is no code dealing with scheduling in the procedures since the scheduling of procedures is performed outside the algorithm layer.

The insertAddress Procedure

Listing 5.4 shows the concurrent and fault-tolerant version of the *insertAddress* procedure. The signature of the procedure has not changed since previous versions, but unlike previous versions, this version actually uses the boolean return value. The procedure will normally return *false* to indicate the caller should not store the contact address, but during crash recovery the crash recovery the procedure might return *true*.

The procedure starts, in lines 10–11, with retrieving the view series of the object and computing the current view of the view series. Since the procedure can be used on an inconsistent tree, the procedure can no longer be sure that the contact field to be modified is either empty or stores contact addresses. The procedure therefore checks, in line 15, whether the contact field already stores a forwarding pointer. If that is the case, this node can only keep the forwarding pointer (line 16) and tell the caller to store the contact

```
procedure insertAddress(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) returns Boolean is
  tentativeCR : ContactRecordView;                  -- tentative contact record
  viewCR : ContactRecord;                           -- initial view of contact record
  storedAddress : Boolean;                          -- address or forwarding pointer
  approved : Boolean;                               -- parent approves address insert
begin
  -- Retrieve the contact record view series and compute the initial view.
  tentativeCR := getTentativeCR(oh);
  viewCR := view tentativeCR;

  -- Add the address or a pointer (possibly temporarily) to the view series.
  -- This change is in main memory only.
  if viewCR(caller).isPtr
    then append view ⟨self(caller).isPtr := true⟩ to tentativeCR;
         storedAddress := false;
    else append view ⟨self(caller).addrSet :=
                        self(caller).addrSet + {addr}⟩ to tentativeCR;
         storedAddress := true;
  end if;

  -- Ask the parent for approval, if needed.
  if $parent \neq NIL$ and (isEmpty(viewCR) or $sizeOf(tentativeCR) > 1$ or
                       inRecovery()) then
    if not isEmpty(viewCR) or doStoreHere(viewCR)
      then approved := call insertPointer(oh, addr, thisNode) at parent;
      else approved := call insertAddress(oh, addr, thisNode) at parent;
    end if;
  else
    approved := true;                               -- No approval needed.
  end if;

  -- Undo the local modification if the parent does not approve, and return
  -- the appropriate response to the child node.
  if approved
    then apply view to tentativeCR;                 -- Make the change persistent
         return not storedAddress;
    else remove view from tentativeCR;
         return false;
  end if;
end insertAddress;
```

Listing 5.4: The fault-tolerant *insertAddress* procedure.

address itself (line 17). Note that a leaf node will therefore never return *true* to a client since it will never refuse to store a contact address. Normally, the contact address will be appended to the view series, in lines 18–20.

Lines 24–25 determine whether the procedure should contact the parent node. The condition the procedure uses is an extended version of the conditions found in previous versions of this procedure. In this case the procedure contacts the parent node if the parent exists and one of three cases applies. The first case is if the contact record is empty. If it is empty, the contact record at the parent is obviously also empty. The second case is if the view series has more than one view appended to it. This means that there are concurrently running update procedures, and the current view of the view series is only tentative. The third case is if the node is in the recovery phase.

In line 26 the procedure decides what RPC call to make. If this node wants or needs to store the address itself, it sends an `insertPointer` request, otherwise it sends an `insertAddress` request. Either way, the approval of the parent node to store the contact address is saved in the *approved* variable. If no RPC is performed, the variable is set to *true*.

Based on the variable *approved*, the *insertAddress* procedure makes its tentative change authoritative by applying it to the underlying contact record (line 37), or it discards the change by removing it from the view series (line 39). In the former case, the procedure returns *true* if a forwarding pointer was stored and *false* otherwise. In the latter case, the procedure always returns *false*.

The insertPointer Procedure

Listing 5.5 shows the concurrent and fault-tolerant version of the *insertPointer* procedure. This version of the procedure follows the same general structure as previous versions. The procedure starts, in lines 10–11, with retrieving the view series for the object and computing its current view. Using the current view, the procedure decides whether it wants to store the contact address or a forwarding pointer, in lines 17–21. The change is appended to the view series in lines 17–18 or line 20.

The procedure then decides, in lines 25–26, whether it should contact the parent node to ask its permission for the change appended to the view series. As for the *insertAddress* procedure, this condition is that the parent exists, and either the contact record is empty, or there are concurrently executing procedures, or the node is in its crash recovery phase. The actual invocation is performed in line 27. The *approved* variable stores whether the parent approves our appended change. If no RPC call is made, it is set to *true*.

Based on the variable *approved*, the *insertPointer* procedure applies its tentative change to the underlying contact record, in line 34, or removes it from the view series, in line 36. In the former case, the procedure returns *true* if a stored forwarding pointer was stored and *false* otherwise. In the latter case, the procedure always returns *false*.

```
procedure insertPointer(oh : ObjectHandle;
                        addr : ContactAddress;
                        caller : NodeID) returns Boolean is
  tentativeCR : ContactRecordView;            -- tentative contact record
  viewCR : ContactRecord;                     -- initial view of contact record
  storedAddress : Boolean;                    -- address or forwarding pointer
  approved : Boolean;                         -- parent approves address insert
begin
  -- Retrieve the contact record view series and compute the initial view.
  tentativeCR := getTentativeCR(oh);
  viewCR := view tentativeCR;

  -- Add the address or a pointer (possibly temporarily) to the view series.
  -- This change is in main memory only.
  if viewCR(caller).addrSet $\neq \emptyset$ or (not viewCR(caller).isPtr and
                                   doStoreHere(tentativeCR))
     then append view ⟨self(caller).addrSet :=
                            self(caller).addrSet + {addr}⟩ to tentativeCR;
          storedAddress := true;
      else append view ⟨self(caller).isPtr := true⟩ to tentativeCR;
          storedAddress := false;
  end if;

  -- Ask the parent for approval, if needed.
  if parent $\neq$ NIL and (isEmpty(viewCR) or sizeOf(tentativeCR) $> 1$ or
                    inRecovery())
     then approved := call insertPointer(oh, addr, thisNode) at parent;
      else approved := true;                  -- No approval needed
  end if;

  -- Undo the local modification if the parent does not approve, and return
  -- the appropriate response to the child node.
  if approved
     then apply view to tentativeCR;          -- Make the change persistent
          return not storedAddress;
      else remove view from tentativeCR;
          return false;
  end if;
end insertPointer;
```

Listing 5.5: The fault-tolerant *insertPointer* procedure.

5.5.3 Delete Operation

Listing 5.6 shows the concurrent and fault-tolerant version of the *deleteAddress* procedure. In contrast to the *deleteAddress* procedure from Chapter 2, this version has an additional boolean parameter, called *delPtr*. The parameter tells the procedure whether it should remove forwarding pointers. This parameter is set to *true* by the calling procedure at the child node if the child's contact record became empty.

The *deleteAddress* procedure starts, in line 11–12, with retrieving the view series for the object and computing its current view. Using the view, the procedure determines whether it can delete a contact address (line 13) or a forwarding pointer (line 14). Note that a forwarding pointer can be deleted only if the contact record at the caller was empty. If something can be deleted (line 16), the procedure will do so in lines 17–37.

The *deleteAddress* procedure continues with determining whether it should remove a contact address or a forwarding pointer, in line 19. The actual deletion consists of appending a new view with the contact address (lines 20–21) or forwarding pointer (line 23) removed from the contact record. To decide what to do next, a new view of the view series is computed, in line 26. If a parent node exists (line 29), the procedure might need to contact it. If the contact record has become empty (line 30), the parent node needs to delete its forwarding pointer (line 31), as indicated by the *delPtr* parameter. If the contact record was nonempty but there were other concurrent procedures or the node was in its recovery phase (line 32), the *deleteAddress* procedure needs to be invoked at the parent as well, but without the possibility to remove forwarding pointers (line 33). Finally, in line 37, when the RPC at the parent has returned, the appended view is applied to the contact record to make the change authoritative.

If no contact address or forwarding pointer was found, the procedure has to check one last case. If the parent node exists and the contact record is empty (line 38), or there is a concurrently executing procedure (line 38), or the node is in its recovery phase (line 39), the *deleteAddress* procedure is invoked at the parent to search for the contact address there (line 40). Otherwise, the procedure is ready.

5.5.4 Move-down Operation

The move-down operation moves a single contact address from a parent node to its child. The operation consists of the *reInsertAddress* procedure executing at the child and the *reInsertPointer* procedure executing at the parent, as described in Chapter 4.

Listing 5.7 shows the fault-tolerant *reInsertAddress* procedure. As with all procedures in this chapter, the *reInsertAddress* procedure starts with obtaining the view series of the object and computing its current view, in lines 9–10. The procedure then determines the contact field in which the contact address has to be inserted, in line 11. If the contact field already stores a forwarding pointer, the address cannot be inserted. If the contact field already stores the contact address, the address does not need to be inserted. In both cases, the procedure is finished, in line 15.

In lines 19–20, the contact address is inserted by appending a new view to the contact

```
procedure deleteAddress(oh : ObjectHandle;
                        addr : ContactAddress;
                        delPtr : Boolean;
                        caller : NodeID) is
  tentativeCR : ContactRecordView;              -- tentative contact record
  viewCR : ContactRecord;                       -- current view of contact record
  addrFound : Boolean;                          -- Can we delete an address?
  ptrFound : Boolean;                           -- Can we delete a pointer?
begin
  -- Retrieve the contact record view series and compute the initial view.
  tentativeCR := getTentativeCR(oh);
  viewCR := view tentativeCR;
  addrFound := addr $\in$ viewCR(caller).addrSet;
  ptrFound := delPtr and viewCR(caller).isPtr;

  if addrFound or ptrFound then
    -- Delete the address or pointer by appending a tentative change.
    -- This change is in main memory only.
    if addrFound then
      append view $\langle$self(caller).addrSet :=
                        self(caller).addrSet $-$ {addr}$\rangle$ to tentativeCR;
    else
      append view $\langle$self(caller).isPtr := false$\rangle$ to tentativeCR;
    end if;

    viewCR := view tentativeCR;                 -- Compute the new current view

    -- Tell the parent to delete its address or forwarding pointer, if needed.
    if parent $\neq$ NIL then
      if isEmpty(viewCR) then
        call deleteAddress(oh, addr, true, thisNode) at parent;
      elsif sizeOf(tentativeCR) $>$ 1 or inRecovery() then
        call deleteAddress(oh, addr, false, thisNode) at parent;
      end if
    end if;

    apply view to tentativeCR;                  -- Make the change persistent
  elsif parent $\neq$ NIL and (isEmpty(viewCR) or sizeOf(tentativeCR) $>$ 0 or
                               inRecovery()) then
    call deleteAddress(oh, addr, false, thisNode) at parent;
  end if;
end deleteAddress;
```

Listing 5.6: The fault-tolerant *deleteAddress* procedure.

```
procedure reInsertAddress(oh : ObjectHandle,
                          addr : ContactAddress) is
  tentativeCR : ContactRecordView;               -- tentative contact record
  viewCR : ContactRecord;                    -- initial view of contact record
  child : NodeID;                      -- Identity of the contact field to use
  success : Boolean;
begin
  -- Retrieve the contact record view series and compute the initial view.
  tentativeCR := getTentativeCR(oh);
  viewCR := view tentativeCR;
  child := determineField(addr);

  -- Determine if we can and need to insert the contact address
  if viewCR(child).isPtr or $addr \in viewCR(child).addrSet$ then
    return;
  end if;

  -- Insert the contact address by appending a tentative change.
  append view ⟨self(caller).addrSet :=
                 self(caller).addrSet + {addr}⟩ to tentativeCR;

  -- Tell the parent to insert a forwarding pointer.
  success := call reInsertPointer(oh, addr, thisNode) at parent;

  -- Undo the local modification if the address was deleted at the parent.
  if success
    then apply view to tentativeCR;                -- Make the update persistent
     else remove view from tentativeCR;
  end if;
end reInsertAddress;
```

Listing 5.7: The fault-tolerant *reInsertAddress* procedure.

```
(1)  procedure reInsertPointer(oh : ObjectHandle,
(2)                            addr : ContactAddress,
(3)                            caller : NodeID) return Boolean is
(4)  begin
(5)    -- Retrieve the contact record view series and compute the initial view.
(6)    tentativeCR := getTentativeCR(oh);
(7)    viewCR := view tentativeCR;
(8)
(9)    if sizeOf(tentativeCR) = 0 and viewCR(caller).addrSet = {addr} then
(10)     -- Exchange the address for a pointer by appending two tentative changes.
(11)     append view ⟨self(caller).addrSet := ∅;
(12)                  self(caller).isPtr := true⟩ to tentativeCR;
(13)     apply view to tentativeCR;                -- Apply the changes directly.
(14)
(15)     return true;                              -- The exchange was successful.
(16)   elsif sizeOf(tentativeCR) = 0 and viewCR(caller).isPtr then
(17)     return true;                        -- Already inserted a forwarding pointer.
(18)   else
(19)     return false;                             -- The exchange could not be done.
(20)   end if;
(21) end reInsertPointer;
```

Listing 5.8: The fault-tolerant *reInsertPointer* procedure.

record. The child node then invokes the *reInsertPointer* procedure at its parent to swap the contact address for a forwarding pointer, in line 23. The result of this RPC call is stored in the variable *success*. If the procedure at the parent was successful (line 26), the appended view is made permanent, in line 27, otherwise the appended view is discarded, in line 28.

Listing 5.8 shows the fault-tolerant *reInsertPointer* procedure. The procedure starts with obtaining the view series of the object and computing its current view, in lines 6–7. Unlike other procedures, this procedure does not perform an RPC. As a result, the procedure appends a tentative change and applies it straightaway. This presents, however, a scheduling problem when there are concurrently executing update procedures for this object. Since these procedures have already appended tentative changes to the view series, the change appended by the *reInsertPointer* procedure at the back of the queue will not be the same change that is made permanent at the head. We therefore ensure that *reInsertPointer* procedure performs its function only when no other concurrently executing procedures exists (i.e., when the number of views is zero), and aborts the move-down operation otherwise.

To allow the exchange of the contact address for the forwarding pointer, the address also has to be the only address in the contact field (line 9). If these two conditions are met, the address is deleted from the contact field and the forwarding pointer is appended in one view, in lines 11–12. This view is applied in line 13, and the procedure signals the caller the exchange was successful, in line 15. If the contact record contains an authoritative forwarding pointer (line 16), the procedure was already executed successfully before. This can happen when either the parent or the child has crashed recently. To provide an idempotent procedure, only the successful completion of the procedure needs to be reported back to the caller, in line 17. Line 18 covers the remaining case where a problem arose and the procedure needs to signal the caller the exchange has failed, in line 19, in which case the whole move-down operation fails.

Chapter 6

Security

This chapter describes our security goals in the location service, and the techniques we employ to achieve them. We specifically focus on threats against the availability of the location service. While in the previous chapter we looked at accidental disruption of availability, here we look at intentional disruption. To achieve a high level of confidence in the security of the location service, we keep our methods as simple as possible and use only well-known techniques.

To simplify the security design, we assume that the location service is operated by a single organization in which all parts of the organization can trust each other. We focus only on mechanisms that allow the location service to protect itself against outside threats and do not worry about one part of the location service trying to attack another part. Research into maintaining a large, trusted, virtual organization across a wide-area network, such as the Internet, is a generic security problem that falls outside the scope of this research.

6.1 Goal

As stated in previous chapters, the location service is central to all communication in the Globe distributed system. It is therefore also the prime target for **denial-of-service** attacks. By disrupting the availability of the location service, an attacker can prevent clients from contacting objects, resulting eventually in a breakdown of the Globe system. There are two ways to attack the availability of the location service. The first attack is to disrupt update and lookup operations in progress. The second attack is to corrupt the location data stored in the location service.

A simple denial-of-service attack that targets running operations is to disrupt all communication in the location service. The main idea behind this kind of attack is to allocate such a large amount of communication resources for useless communication that no resources are left for useful communication. For instance, the process of setting up a connection through a three-way handshake requires some state at the accepting side of the connection. By sending a large number of connection requests in parallel, an attacker can tie up all memory resources dedicated to setting up connections. This type of attack has been shown to be effective against TCP/IP [Schuba et al., 1997].

A second way to disrupt operations in progress is by generating a large number of valid operations. For example, an attacker can continuously request insert and delete operations or search for all contact addresses of a large number of objects. The amount of resources used to handle these valid but otherwise useless requests, prevents ordinary requests from being handled in a timely fashion, resulting in, for instance, time-outs and aborted lookup operations.

The general protection mechanism against these types of denial-of-service attacks is **resource accounting** [Leiwo et al., 2000]. By recording the amount of resources used by a client and placing a limit on that amount, we can prevent clients from overloading a physical node. An important part of this accounting process is the authentication of clients. Unfortunately, in most cases significant amounts of resources are needed before the identity of a client can be securely known. For instance, when setting up a communication channel, the identity of the peer is not yet securely known, and the authentication protocols used to establish the identity of the peer are resource intensive as well.

To prevent denial-of-service attacks in cases where a client's identity cannot be securely determined, different researchers have proposed solutions based on **puzzle solving** [Leiwo et al., 2000]. In these schemes, a server that fears it is under a denial-of-service attack, refuses to allocate resources to a client before the client has solved a resource-intensive puzzle. The basic notion is that a legitimate client can commit the resources needed to solve a single puzzle, but that an attacker cannot commit the resources needed to solve enough puzzles to overload the server with useless connections. For these schemes to work, the generation of puzzles and the verification of their solutions should require only a minimal amount of resources from a server. After a client provides the correct solution to the given puzzle, it is allowed to setup a connection, authenticate itself, and use the location service.

Our focus in this chapter is on preventing denial-of-service attacks against objects through the unauthorized modification (i.e., corruption) of data in the location service. We specifically want to prevent the unauthorized deletion of valid contact addresses and the unauthorized insertion of invalid contact addresses through normal update operations. If any client can delete contact addresses of any other client, all contact addresses run the risk of being removed by an attacker. Furthermore, inserting large numbers of invalid contact addresses for an existing object will make a valid contact address of the object virtually invisible to clients.

Access control on update operations is the main form of protection against denial-of-service attacks that target the data stored in the location service. The location service has to determine which changes to a contact record are allowed and which are not. Since the location service cannot determine by itself which update operations are allowed, clients will have to determine the correct update policies for their objects.

Access control on update operations requires us to solve four problems:

- Secure communication between clients and the location service.
- Secure communication within the location service.
- Client authentication by the location service.
- Determining the update policy for an object and granting access accordingly.

To correctly enforce the update policies, the access control mechanism requires both the integrity of communication channels to ensure that update requests are authentic and intact, and the authentication of clients. Luckily, these two issues are well-known security problems with good solutions. Providing a scalable mechanism for per-object update policies is a new problem, and the description of our solution therefore covers a major part of this chapter.

Since the lookup operation does not change anything, it is generally available to all clients, as long as a client is allowed to use the resources needed for the operation. The location service currently does not support confidentiality of location information since confidentiality is useful only when it is supported throughout the Globe distributed system. Given that object handles and contact addresses are not kept confidential within Globe, it is useless to keep them confidential within the location service.

However, we could have chosen to support the confidentiality of the set of contact address associated with an object handle. For instance, a reason to keep the set confidential is to provide a first line of defense against denial-of-service attacks on objects. Since providing confidentiality would simply require access control for lookup operations and encrypted communication within the location service, we are confident we can easily add such functionality should the need arise.

It is important to realize that the location service is not part of the access control system that protects the integrity of objects. The location service is involved only in finding a replica of an object. We assume that the Globe system provides end-to-end checks between clients and objects during the binding process, authenticating both the client to the object and the object to the client. Afterward, an object enforces its own access policy on the method invocations it receives.

6.2 Object Model

We introduce an object ownership model to determine who is allowed to make which changes to the set of contact addresses of an object. In our model every object has a single location-service client, called the **object owner**, that ultimately determines the set of contact addresses of an object. For a (highly) replicated object it is undesirable, however, that there is only a single entity that can perform update operations on the set of contact addresses. To ensure locality, the object owner needs the ability to **delegate** (restricted) update rights to other clients of the location service. These clients can then independently of the object owner insert and delete contact addresses at their local leaf node, for example, when they create or remove replicas.

To keep some control over the set of contact addresses, the object owner should not give these other clients unrestricted access to the set. Instead, the owner should determine an **update policy** for its object that specifically states which update rights are delegated to these clients. The location service should be able to read the policy and validate specific update requests. An **update right** in this model allows a client to insert contact addresses with certain characteristics and to delete contact addresses with certain characteristics. These two rights are combined since it makes no sense to separate them and give a client the right to insert a contact address without the right to delete it (or vice versa).

The update policy limits the basic update right by describing the characteristics of the update operations it allows. Currently, the location service supports three characteristics: the time and duration the contact address is to be stored, the location (e.g., network or geography) of the contact address, and the property maps of the contact address. For example, the update policy of an object might say that a specific client can insert contact addresses of slave replicas (a property map restriction) during a week (a time restriction) at the Amsterdam leaf node (a geography restriction). The three characteristics are not the only ones possible. Other characteristics can also be supported should the need arise.

The time characteristic also influences the duration the contact address is stored in the location service. As stated in Chapter 2, contact addresses are stored using a lease. When the lease runs out, the contact address is deleted by the location service. Since the time characteristic limits the period a client can have contact addresses in the location service, this characteristic also limits the lease time of a stored contact address.

If an object is highly replicated, the process of managing and distributing update rights to other clients might become too large for the object owner to handle on its own. An object owner therefore also needs the ability to delegate the **delegation right**, that is, allow another (trusted) client to delegate to other clients update rights and possibly the delegation right itself. While this recursive step can be repeated indefinitely, we expect only a few delegation steps to be useful. The object owner should therefore be able to limit the (recursive) delegation right to a small number of steps. For instance, an object owner can allow a trusted client to delegate the update rights to other clients, but refuse the trusted client from delegating the delegation right itself.

As a general policy, a client that delegates update rights is allowed to delete all contact addresses that were inserted using the update rights it has delegated. This policy is needed to allow a client to remove invalid contact addresses inserted by a misbehaving client to which it had previously delegated update rights. As a consequence, this policy also gives the object owner full control over all the contact addresses of the object that are stored in the location service since all update rights are always given (possibly indirectly) by the object owner.

Figure 6.1 shows, as an example, the delegation relationships between the object owner and eight other clients that can perform update operations. Beside the object owner, only clients $\mathtt{C}_1$ and $\mathtt{C}_2$ have both update and delegation rights. Clients $\mathtt{C}_3$ to $\mathtt{C}_8$ have only update rights. Contact addresses inserted by client $\mathtt{C}_3$ can be deleted by client $\mathtt{C}_3$ and the object owner; contact addresses inserted by clients $\mathtt{C}_6$ to $\mathtt{C}_8$ can be deleted by the object owner, client $\mathtt{C}_2$, and the client that inserted the contact address. The figure does not show

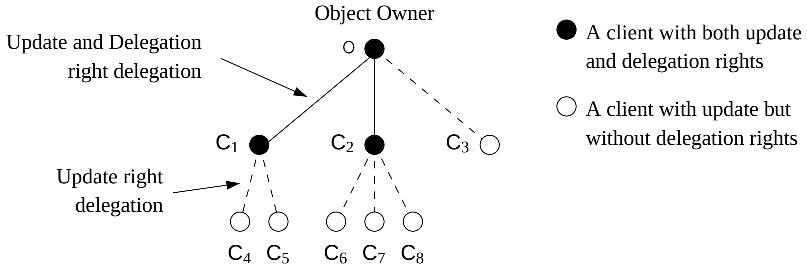

Object Owner
O
Update and Delegation right delegation
C1
C2
C3
Update right delegation
C4
C5
C6
C7
C8
● A client with both update and delegation rights
○ A client with update but without delegation rights

Figure 6.1: Update and delegation right delegation.

whether clients $\mathtt{C}_1$ and $\mathtt{C}_2$ are allowed to delegate the delegation right.

6.3 Secure Communication

To ensure that the access control mechanism works, we need to protect the underlying communication system. We specifically need to ensure that a physical node can distinguish valid RPC requests and replies from those sent or modified by an attacker. Otherwise, an attacker can simply trick a physical node into performing unauthorized changes by impersonating another physical node that is authorized to make changes. This problem can be avoided by letting the sending node sign its request or reply cryptographically, and letting the receiving node verify the signature on the request or reply. Since all location information is public knowledge, no encryption is needed to ensure confidential communication.

We protect the integrity of the communication channels between physical nodes using **public-key cryptography** (PKC) [Diffie and Hellman, 1976]. Every physical node has a key pair consisting of a public and a private key. Every physical node signs its RPC messages with its private key, and other nodes verify these messages with the node's public key. Since the exact public key algorithm used is irrelevant to our discussion, any applicable PKC algorithm can be used, for instance, RSA [Rivest et al., 1978].

We choose PKC over **shared-key cryptography** (SKC), also known as secret key cryptography, because in our specific situation PKC is more scalable. The main problem with an SKC-only system is its dependence on a trusted third party for generating, storing, and handing out shared keys for every pair of nodes in the system. Since every physical node should be able to communicate with every other physical node, the implementation of the trusted third party will require a significant infrastructure and can easily become a communication bottleneck in the location service. Furthermore, when SKC would be used in a wide-area system, such as our location service, the interaction with the trusted third party would require long-distance communication, violating our locality principle.

An important issue in any system using PKC is the secure distribution of public keys. Since physical nodes use these keys to authenticate RPC messages, they have to be sure the public key used to authenticate a message belongs to the sending physical node. The component responsible for the secure distribution of public keys in a PKC system is called the **public-key infrastructure** (PKI). The PKI consists of certification authorities (CA) that create certificates by signing (physical node identifier, public key)-pairs. A physical node accepts the authority of the CA and trusts the binding of the public key to the node, as stated in the certificate. The certificates then only need to be available to interested physical nodes.

For secure communication between physical nodes in the location service, we make use of a dedicated PKI. Adding a dedicated (internal) PKI to the location service is straightforward since we already have a **tree information service**. The tree information service provides information about the search tree, such as its logical structure, the partitioning of logical nodes, and the contacting information of physical nodes. It can easily also provide public keys as part of the contacting information. The tree information service is described further in Chapter 7 on tree management issues.

The difference in scalability between the trusted third party of an SKC system and the PKI of a PKC system is that the number of public keys stored in a PKC is much smaller than the number of shared keys in an SKC. The number of keys stored in a PKI is of the same order as the number of physical nodes (i.e., $O(n)$), while the number of (shared) keys stored by the trusted third parties is in the order of the square of the number of physical nodes (i.e., $O(n^2)$). Furthermore, the creation of certificates (i.e., the signing of public keys) can be done off-line, leaving the tree information service with the simple task of distributing certificates.

An important problem with PKC systems is how to deal with key revocation. When the private key of a physical node is compromised (i.e., potentially known by attackers), other physical nodes can no longer use the public key of the pair to verify RPC request and replies. These other nodes therefore need to be informed of the fact that this public key is revoked. Fortunately, given the fact that physical nodes have to contact the tree information service regularly anyway, they can easily receive key revocation information during this contact.

PKC has several problems, most of which can be alleviated by using a hybrid public-key and shared-key system. The first problem is that public key operations are slow (i.e., require many CPU cycles). The usual way to solve this problem is to setup a shared session key using PKC, and then use this shared key to sign request and reply messages. That way the heavyweight public-key operations need to be done only infrequently. The use of session keys in a hybrid system also solves a second problem which is that public keys are easier to break than shared keys of the same size. Since the public key is less frequently used in a hybrid system, an attacker can collect less information to break the key. A third problem is that public keys are larger than shared keys. Whereas the typical size of a shared key is between 128 and 256 bits (16 to 32 bytes), the typical size of a public key is between 1,024 and 2,048 bits (128 to 256 bytes). A public key can thus be said to be on average a factor eight times larger than a shared key.

The size of the public keys of physical nodes can, unfortunately, pose a problem in location caches. Recall that with the introduction of location caches, described in Chap-

Table 6.1: Keys used together with a reconnection ticket.

Symbol	Name	Lifetime	Known by
KL_n	Long-term key	Long	Node *n*
KM_n	Medium-term key	Medium	All nodes in the tree
$KS_{n,m}$	Session key	Short	Nodes *n* and *m*

ter 4, lookup operations avoid general tree traversal by following a reference from one node in the search tree to another node in the tree that potentially stores a contact address. Following the node reference means communicating (securely) with the physical node that stores the contact record for the object at the referenced logical node. Every physical node thus potentially needs to know the public key of every other physical node in the tree.

The naive way to obtain the public key of a physical node is to query the tree information service when its key is needed. This approach is, however, undesirable since it would slow down and increase the resource usage of the lookup operation considerably. A simple way to avoid the communication with the tree information service is to let the physical node that stored the contact address return a certificate with its public key together with the contact address and its node identifier during a lookup operation. Every node along the return path of the lookup operation can then validate and store this public key in its location cache together with the node identifier. Storing public keys in the location cache leads, unfortunately, to a significant increase in the size of a cache entry given the size of public keys.

Fortunately, the problem of the increased cache entry size can also be diminished by using a shared-key technique. This technique is based on the concept of a **reconnection ticket** and uses three types of shared keys. The characteristics of the three types of keys are shown in Table 6.1. To create a reconnection ticket, we associate a long-term key KL_n and a medium-term key KM_n with every physical node *n*. Physical node *n* keeps its long-term key KL_n a secret, but shares its medium-term key KM_n with other physical nodes in the search tree. However, physical nodes do keep medium-term keys a secret from everyone outside the search tree. A physical node regularly changes its medium-term key to lessen the consequences in case the key is compromised.

Physical node *n* uses its long-term key KL_n and medium-term key KM_n to create a reconnection ticket. The reconnection ticket (shown in Figure 6.2) consists of a plaintext part and an encrypted part. The plaintext part consists of the medium-term key KM_n of physical node *n*. The encrypted part consists of the medium-term key KM_n with some administrative information encrypted with long-term key KL_n. The administrative part, for instance, describes how long the reconnection ticket remains valid. The reconnection ticket is given to other physical nodes in the search tree to enable them to reconnect to node *n* securely at some time in the future. Since the medium-term key KM_n is a shared key and available in plaintext in the ticket, the ticket has to be stored and communicated

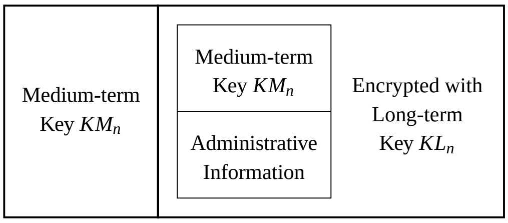


Figure 6.2: The structure of a reconnection ticket.

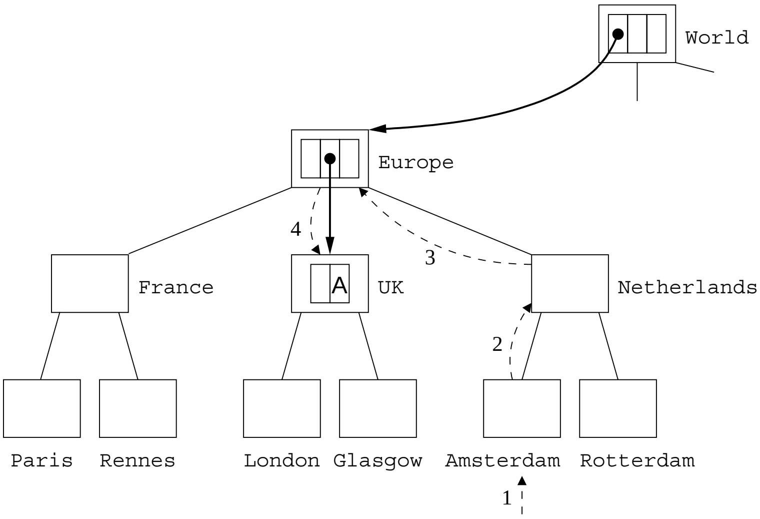


Figure 6.3: In the first half of the first lookup operation, the contact address is found at the `UK` node.

confidentially. We therefore need to encrypt the ticket for secure transmission.

The use of the reconnection ticket in the lookup operation is shown in Figures 6.3–6.5. Since we are dealing with the location cache, we need to consider two lookup operations. The first lookup operation retrieves a node identifier (i.e., a node reference) and a reconnection ticket; the second lookup operation uses the node identifier and the reconnection ticket to efficiently and securely retrieve the contact address a second time.

Figure 6.3 shows the first half (i.e., step 1–4) of the first lookup operation. In the figure, the lookup operation traverses the search tree and finds a contact address at the `UK`. This happens in the same way as described in Chapter 4. Note that this part of the example does not involve the reconnection ticket.

In the second half of the first operation (shown in Figure 6.4), the `UK` node returns

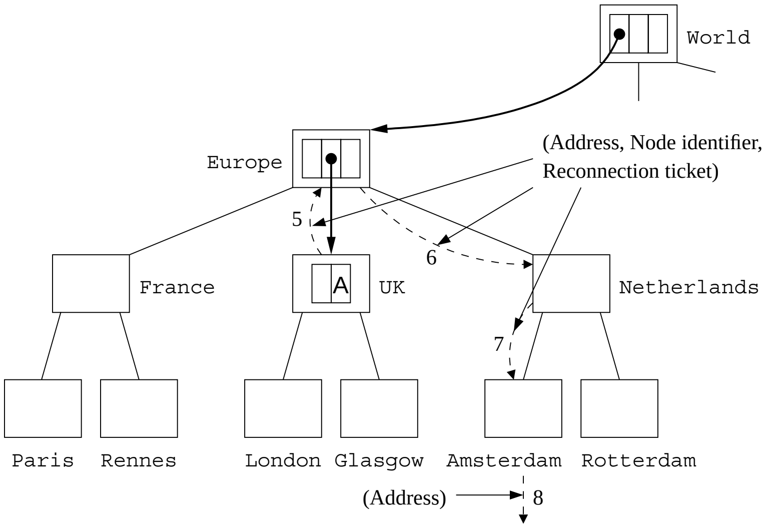


Figure 6.4: In the second half of the first lookup operation, the (contact address, node identifier, reconnection ticket)-triplet is returned from the UK node to the Europe, Netherlands, and Amsterdam nodes. Note that, in step 8, the client receives only the contact address.

its reconnection ticket, together with the contact address and its node identifier, in the reply message of the lookup operation. Every physical node along the return path of the lookup operation (i.e., the Europe, Netherlands, and Amsterdam node) can store the node identifier and the reconnection ticket in its location cache, and use them to securely (re-)connect to the UK node.

Figure 6.5 shows the second lookup operation that uses the cached node identifier and reconnection ticket stored in the Netherlands node. When the (physical) Netherlands node retrieves the entry of the UK node from its location cache, it also retrieves the reconnection ticket of the UK node. It then uses the plaintext medium-term key KM_{UK} from the reconnection ticket to encrypt a new session key $KS_{NL,UK}$, and sends its lookup request signed with $KS_{NL,UK}$ to the UK node, as shown in step 3 of Figure 6.5. This request includes the session key $KS_{NL,UK}$ encrypted with the medium-term key KM_{UK} and the encrypted part of the reconnection ticket, that is, the medium-term key KM_{UK} and the administrative information encrypted with the long-term key KL_{UK}.

Since the UK node can decrypt the encrypted medium-term key using its own long-term key KL_{UK}, it knows that it has given its medium-term key KM_{UK} to the Netherlands node (possibly indirectly) and that the Netherlands node is thus trustworthy. Furthermore, using the medium-term key, it decrypts the session key $KS_{NL,UK}$. It uses the session key to sign its lookup reply, sent back in step 4. Using the signature, the Netherlands node can verify the validity of the reply message. Reconnection tickets

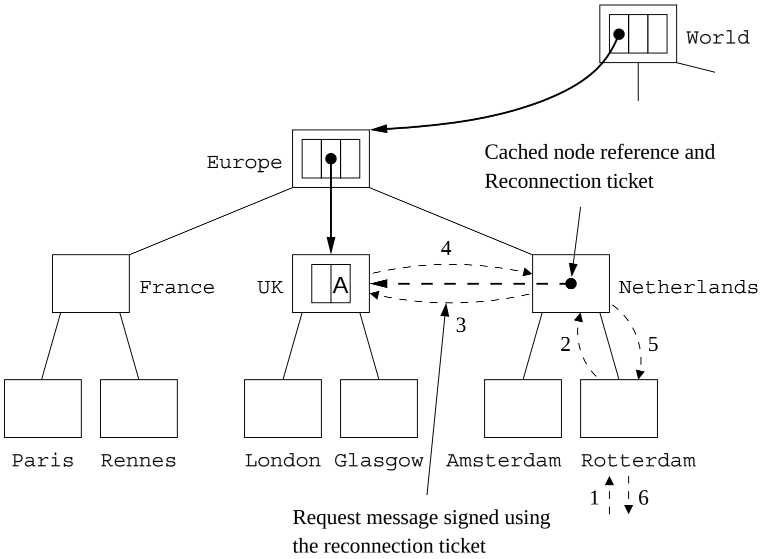


Figure 6.5: During the second lookup operation, the Netherlands node uses the node reference and reconnection ticket from its location cache, obtained in Figure 6.4, to communicate securely with the UK node and obtain the contact address.

can also be used to provide a fast secure connection setup in other places, for instance, between pairs of physical nodes in a logical parent and child. The reconnection ticket method is similar to an approach for DNS proposed by Ateniese and Mangard [Ateniese and Mangard, 2001].

Communication between clients and leaf nodes also needs to be secure. Leaf nodes need to authenticate clients for the resource accounting described in the previous section, and clients need to be sure they communicate with the location service. The algorithms and protocols used by a leaf node to authenticate a client and protect communication can be different at different leaf domains. For instance, a shared-key system, such as Kerberos [Steiner et al., 1988], can be used in one leaf domain and a public-key system in another. Either way, the client authentication system is independent of the internal authentication system of the location service. Allowing leaf domains their own security infrastructure increases the flexibility of the location service. It does, unfortunately, also increase the trusted computing base (TCB), and thereby decreases the security of the location service. Client authentication for access control is described in Section 6.6.

6.4 Object Ownership

The ownership model requires that the location service is able to determine that an update request is authorized, that is, determine that the request either came from the object owner or from a client that was authorized by the object owner to make changes. To make this determination, the location service needs to be able to find out who the owner of an object is. In the location service, object ownership is established using public-key cryptography. Every object has a public key associated with its object identifier, and only the object owner can prove its ownership by proving it has knowledge of the corresponding private key.

The public key is associated with the object identifier using a separate **ownership service**. To increase security, the client uses a different key pair for each object. When a client creates a new object, it creates a new key pair and gives the public key from the pair to the ownership service. The service returns an object handle that contains an association between the object identifier and the public key. The service also uses other information to create the object handle, such as the expected home location of the object (see Chapter 3). Since new objects will be created regularly all over the world, the ownership service needs to consist of multiple servers distributed over the world to provide local access.

The association of a public key with an object identifier can be implemented in several ways. Figure 6.6 shows four ways to implement the association: (a) using an external database to store the public key, (b) using the public key as the object identifier, (c) using a certificate, and (d) deriving the object identifier from the public key. In some of these methods the public key is part of the object handle; in others the public key is obtained separately, for instance, it is provided by the client as part of the update request.

In the rest of this section, we compare the four methods of Figure 6.6 based on security and performance issues. We are, for instance, interested whether the association is static or dynamic, what the vulnerabilities and their consequences are, whether public key retrieval and object handle creation are local operations or not, and what the impact is of the method on the size of data structures, such as database entries. The latter is particularly important, given the size of public keys. The actual method used in the location service is irrelevant for the rest of this chapter, but the method based on an object identifier derived from a public key is the most attractive one currently.

Public Key Stored in Database

The simplest way to implement the association between an object identifier and the public key is to store the public key in a database at the ownership service and allow location service nodes to query this database. Figure 6.6(a) shows this implementation with the object handle consisting of an object identifier and location information. The public key is retrieved during update operations when needed. There are two variants of this implementation.

In the first variant the public key is retrieved by the leaf node at the start of an update operation. This scheme has several desirable security properties. The association between

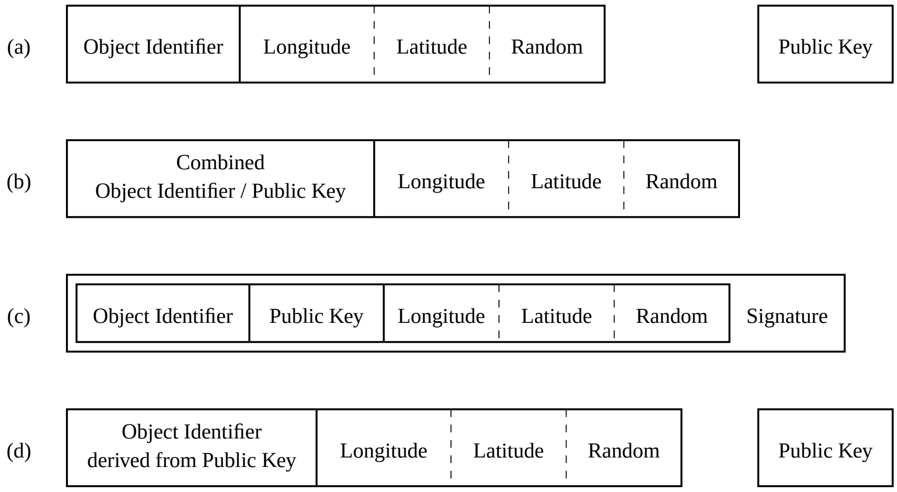


Figure 6.6: Four methods of (securely) associating ownership information with an object identifier: (a) using an external database to store the public key, (b) using the public key as the object identifier, (c) using a certificate, and (d) deriving the object identifier from the public key.

object identifier and public key is dynamic, that is, when desired the owner can change the public key. Since the association is maintained by separately administered servers, changes are made in one place and the service can easily provide audit logs of changes made to associations. Unfortunately, this variant has a severe scalability problem. Since clients can request update operations from all over the world, the ownership service will need to communicate over long distances, violating locality.

The second variant of the database implementation improves the locality of the first variant. In the second variant the public key of an object is also stored in the location service itself. This means that if the object is already known by the location service, no communication is needed with the ownership service. When an object is unknown, the update operation will reach the root node, which will then retrieve the public key from the ownership service, and forward it to rest of the nodes that store contact records. Since the (physical) root node is stored in a single place, it is possible to co-locate the ownership server, and provide local communication between the physical root node and ownership server.

Unfortunately, by also storing the public key in the contact records, we lose the dynamic association property of the first variant. Since there is no longer a single place where the association is stored, it can no longer be changed easily. A different problem with storing the public key in the contact record is that it also increases the size of the contact record.

In both variants, there are two threats to the system. The first threat is an attacker that gains access to the database and changes its contents. The second threat is against the communication between the ownership service and the location service. If an attacker manages to impersonate the ownership service it can provide any public key it wants, and thus also impersonate the object owner. Both threats can easily be counteracted, however, by standard security technology, such as authenticated communication.

Public Key Used as Object Identifier

The main problem with the database method is the communication needed to retrieve the public key. This problem can be overcome by using the public key itself as the object identifier, as shown in Figure 6.6(b). The reason we can use the public key as an object identifier is that for the public key to be secure, it has to have a very high probability of uniqueness. Thus, if we generate a new public key for every new object, we can be sure that the public key is not currently associated with another object. Furthermore, we can be sure this public key was never associated with an object at all. If we combine this with the fact that we only associate a single public key with an object, the public key becomes a proper identifier [Wieringa and de Jonge, 1995], which is a requirement of object handles (see also Section 1.5). The Legion system uses a similar approach to secure object-to-object communication [Stoker et al., 2001].

The main advantage of this approach is its simplicity. No extra mechanisms are needed to associate the object identifier with public key. A separate ownership service is therefore not needed in this scheme. The functionality needed to combine the public key and the location information into an object handle can simply be provided by a library function. This scheme is thus completely decentralized.

The main drawback to this approach is that it increases the size of the object handle. If we assume that the object identifier was originally 128 bits long and that a public key is on average 1,024 bits long, the object handle grows by a factor of eight. Since the object identifier is used as the indexing key in the local contact record database at every node, this increases the storage size for the (object handle, contact record)-pair entries in the database.

The only credible risk of this method is the exposure of the private key of individual objects. This can happen only outside the location service since the private key is only stored by the object owner. Fortunately, the consequences of a key exposure are limited to the contact addresses of the compromised object. The situation is rectified by making the object available using a different public key and object handle.

Object Handle as a Certificate

A different way to associate an object identifier with a public key is by using a certificate, as shown in Figure 6.6(c). The object handle is in this case a certificate signed by the ownership service. When the ownership service receives the object's public key and location information, it generates a unique object identifier. It then combines the public key and location information with the object identifier to form an object handle. The ownerships service then signs the object handle using its private key, turning the object handle into a certificate.

From a security standpoint, it is unwise to give all the servers of the ownership service the same key pair to sign object handles. Instead, every server should have a private key pair. Leaf nodes of the location service, however, need to know the server's public key when authenticating the object handle. The client should therefore provide the public key of the ownership server as a separate certificate beside the object handle. These certificates will be signed by the ownership service certificate authority.

The main advantage of this approach is that the public key can be locally obtained and verified. Furthermore, once the public key is used to validate the request, the location service can forget the public key and certificate information, leaving a much smaller object handle consisting of only the object identifier and location information. A small advantage of this approach is that the certificate also securely associates the location information with the object identifier. This means that object handles can be passed through insecure channels without the risk of being changed.

The main problem with the certificate implementation is the risk of a key exposure of an ownership server. If the private key of a server becomes known to some attacker, the attacker can associate its own public key with an object identifier, and thereby perform unauthorized update operations on that object's handle. The attacker can then change the set of contact addresses of any object. To deal with this situation, the ownership service needs a mechanism to quickly distribute certificate revocation lists to location service nodes.

Object Identifier Derived from Public Key

Instead of using the public key directly, we can also implement the association by using an object identifier that is securely derived from the public key. In this implementation the object identifier is derived from the public key using a secure message digest, such as SHA-1 [National Institute of Standards, 1995]. This method is shown in Figure 6.6(d). Since it is exceedingly difficult to find two public key pairs with public keys that have the same message digest, the public key is securely associated with the object identifier. A similar approach is used in the SFS distributed file system [Mazières and Kaashoek, 1998].

This approach has two advantages. The first advantage of this approach is that the object identifier is less than one sixth the size of the method that uses the public key directly (a public key is 1,024 bits; a SHA-1 message digest size is 160 bits). The second advantage is that both creation of an object handle and the validation of the association between public key and object identifier are local processes. This method is therefore, like the direct-use method, completely decentralized. The main disadvantage is the fact that clients need to manage public keys explicitly, that is, in addition to managing object handles.

Like in the direct-use method, the only credible risk with this method is the exposure of the private key of single objects. Fortunately, the consequences are limited to the contact addresses in the location service of the object. The situation is rectified by making the object available using a different public key and object handle.

6.5 Update Policy

The update policy for an object is implemented using a set of **update credentials**. Every client authorized to make changes to the set of contact addresses of an object is provided with a personal update credential by the object owner or one of its delegates. The client gives this credential to the leaf node with its update request to prove it is allowed to make changes. Update credentials are very similar to the credentials used in Legion [Stoker et al., 2001].

An update credential consists of an attribute certificate containing general information, such as the object handle, and a chain of attribute certificates with every certificate providing proof that a client has certain update and delegation rights. The main information currently stored in the general information certificate is the home location in the object handle. The reason for providing this information is to ensure that the correct home location is used when selecting physical nodes (see Chapter 3). The chain of certificates starts with a number of certificates allowing delegation steps and ends with a certificate that binds the client's public key and the allowed characteristics of update requests together. The object owner provides an empty update credential to signal it is itself performing the update operation.

Figure 6.7 shows an example update credential for location service client 3. The update credential consists of four attribute certificates. The leftmost certificate contains the object handle of the object, and, in the future, possibly other information relevant to an update operation. The other three certificates form the certificate chain. The first (leftmost) certificate of the chain is signed by the object owner using the public key of the object. It thus needs to be verified with the public key of the object owner. The certificate grants client 1 the right to delegate update rights for the object by including the public key of client 1 (i.e., "P. Key: <key 1>") in the certificate. This delegation right is valid from the date 5/4 for 28 days. Furthermore, client 1 can delegate update rights only for the specified geographical area ($5^\circ \pm 3^\circ$ longitude by $52^\circ \pm 3^\circ$ latitude) and for the specified property map (0011). Client 1 used its right to generate the next certificate in the chain, providing client 2 with a more restricted delegation right. Client 2 used this right to provide client 3 with the right to perform update operations on the object. Note that the restrictions in the certificates increase with every step in the certificate chain.

6.6 Access Control

Access control consists of two steps. The first step is performed at the leaf node; the second step is performed at the node that stores or will store the contact address. The leaf node is responsible for authenticating the client and checking the update credential.

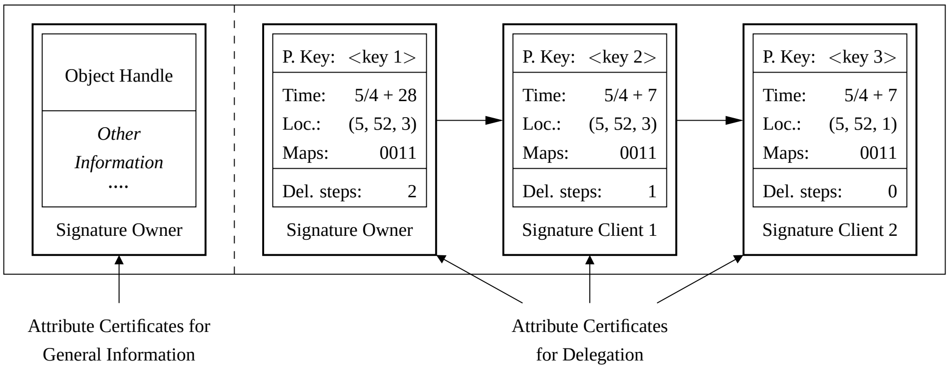


Figure 6.7: An example update credential for client 3, consisting of one attribute certificate for general information and three attribute certificates for update-right delegation steps. Note that the "Signature Owner" is a signature made by the object owner using the public key of the object.

The node that stores the contact address is responsible for checking whether the client is allowed to perform the update operation (primarily the delete operation) and storing the information needed to perform this check.

When a leaf node receives an update request, it first extracts the parameters: object handle, contact address, property map, and update credential. The leaf node then extracts the public key from the object handle using one of the methods described in the previous section. Using the public key of the object, the leaf node is now able to authenticate and validate the update credential. Authenticating the update credential consists of validating the signature on the attribute certificate for the general information and on every attribute certificate in the certificate chain. Validating the credential consists of ensuring that the semantics of the characteristics in the chain are not violated, that is, with every delegation step the restrictions on the allowed update operations should be the same or stronger. The number of allowed delegation steps should always decrease.

After the update credential is authenticated and validated, the leaf node extracts the client's public key from the last (i.e., rightmost) certificate of the update credential. In the example of Figure 6.7, this is the public key of client 3. The leaf node uses this public key to authenticate the client. This authentication is independent of the authentication for resource accounting purposes described in Section 6.3. The client is authenticated using a challenge/response system. The client authenticates itself by proving it knows the private key of the public key in the credential. For example, the leaf node sends a message to the client containing a number encrypted with the public key (the challenge); the client proves its identity by responding with a message containing this value in plaintext (the response).

Finally, the leaf node ensures that the update requests conforms to the limits set in the update credential. First, the leaf node checks whether the object handle given as a parameter is the same object handle given in the update credential. Afterward, the leaf node checks whether the current time at the leaf falls in the time range of the credential, and uses the time range to set the lease time for the contact address. Finally, the leaf node checks whether it is at the right geographic location, whether the contact address falls in the right address range, and whether the property map of the address matches the property map restrictions. If these checks succeed, the update operation is allowed to proceed in the location service.

To support our policy that allows the clients on the certificate chain (i.e., the clients that delegated the update right) to delete the contact address, the insert operation needs to store access control information with the contact address. This information should allow the location service to identify these other clients. Luckily, this information is stored only with the contact address. Forwarding pointers do not require such information since they are created and removed only in response to the insertion or deletion of contact addresses. If we were to store the public keys of the clients in the update credential, the contact record could become very big. Fortunately, we need to store only the message digests of the public keys, keeping the size of the additional security information small.

The delete operation needs to ensure only that the client invoking the operation is actually allowed to delete the contact address. It performs this access control check at the node where the contact address is stored since this node also stores the message digests of the public keys of all clients that were part of the update credential in the original insert operation. The delete operation needs to verify that the message digest of the client's public key matches a message digest stored with the contact address. Deleting forwarding pointers in the rest of the tree requires no further checks.

Chapter 7

Tree Management

This chapter provides an outlook on the problems relating to the management of the location service. Specifically, it deals with changes in the environment in which the location service operates and how these changes need to be reflected in the configuration of the distributed search tree. Since tree management is at the time of writing still part of ongoing research, this chapter does not provide (complete) solutions to these problems.

7.1 Dealing with Change

The location service is not an autonomous component of the Globe distributed system, instead it needs to be managed by a support organization. This organization needs to ensure that all physical nodes are up and running, that the nodes have all the resources they need, and that clients are allowed to use these resources. The main goal of the organization managing the location service is, however, to ensure the best possible performance level of the location service. As in the previous chapter, we assume in this chapter that a single organization is responsible for managing the location service.

The performance of the location service can be characterized in three ways. First, the location service should provide high-quality answers. This means that a lookup operation should return the contact address(es) nearest to the client that initiated the lookup operation. Second, the location service should ensure that the duration of all operations is as short as possible. This applies especially to the lookup operation since the client wants to use the contact address that is found. Third, the resource usage of the location service should be kept as small as possible since the location service does not provide application-level functionality and its resource usage should thus be considered overhead. In short, the management organization needs to ensure that the location service is *effective* and *efficient*.

The performance of the location service is strongly influenced by the configuration of the distributed search tree. We distinguish two types of configuration information: the logical structure and the physical structure. The logical structure deals with the shape of the search tree, such as the parent-child relationships. The physical structure deals with the implementation of logical nodes, such as the number and placement of physical nodes.

The logical structure of the search tree should be strongly influenced by distances in the underlying network since the location service uses this structure to exploit locality, and thereby provide high-quality results. Recall that the metric for network distance in the location service is based on a combination of geographical, network-topological, and administrative boundaries. The physical structure of the search tree should be determined by the distribution of clients and objects since these greatly influence the distribution of the workload. Ensuring efficient resource usage in the location service means ensuring that the right amount of resources are available at the right locations in a domain.

Unfortunately, neither the network distance nor the client or object population is static. For instance, new network connections are added continuously and existing connections are upgraded. These changes decrease the latency of communication and allow more clients to use the location service, possibly in new locations. Since the network and client environment changes over time, maintaining a high performance level requires regular adaptation of the configuration of the location service to the current environment.

When changing the configuration of the location service, we face several problems. As noted before, we cannot stop the location service without violating its high-availability requirement. Since we cannot stop the location service, at least not as a whole, we need to make the changes to the configuration on-the-fly. We also face our "general" scalability problems. A change in the configuration can potentially affect a large number of physical nodes, for example, in the root domain, and therefore involve a large amount of data. These same physical nodes are potentially distributed worldwide, risking the need for wide-area communication. The methods that change the configuration of the location service should deal with these problems.

We distinguish three types of changes to the configuration information. First, we can change the contact information of a physical node. This will happen, for example, when the network address of a node is reconfigured. This problem will be dealt with in Section 7.2. Second, we can change the physical structure of the search tree by adding physical nodes to or removing them from the set of nodes that implement a logical node. The problems caused by this type of change are dealt with in Section 7.3. Third, we can change the logical structure of the search tree by adding, removing, splitting, or merging logical nodes. The problems caused by this type of change are dealt with in Section 7.4.

The management organization needs to gather statistical information to determine when to make which changes to the location service configuration. The organization needs to monitor the network to ensure that the domain hierarchy still represents the distances in the network. It also needs to monitor the load at the physical nodes to ensure that certain nodes are not overloaded. In this chapter we assume that this information is readily available to determine the appropriate changes.

The organization managing the location service will also have other responsibilities relating to the location service, such as client administration, hardware maintenance, and backup operations. While these issues are relevant to the management of the location service and have to be dealt with, they do not involve changes to the search tree. They fall therefore outside the scope of the research described in this chapter.

7.2 Tree Information Service

Regularly adapting the search tree to the current environment requires the logical and physical search tree structures to be dynamic. Unfortunately, this means that it is no longer obvious for a physical node that has just started what the current logical and physical structure of the search tree is. To solve this problem, a description of the current structure of the search tree is maintained by a separate service, called the **tree information service**. A physical node can query this service when it needs to know the current structure of the search tree, for example, its current logical parent node.

The tree information service stores the configuration of the search tree using two tables. The first table stores **logical node records**; the second table stores **physical node records**. These tables are indexed by the logical and physical node identifier, respectively. A client retrieves a logical or physical node record from the tree information service by providing the identifier of the relevant node.

The logical node record stores the following properties of a logical node:

- The identifier of the logical node itself
- The identifier of the logical parent node
- A list of identifiers of the logical child nodes
- A list of identifiers of the physical nodes implementing the logical node
- The mapping table of the logical node
- A version number

To identify the logical node described by the logical node record, the record starts with the identifier of the logical node. To store the logical tree structure, every logical node record stores the identifier of the logical parent node and a list of the identifiers of the logical child nodes. A special `NIL_NODE` value is used to indicate a node does not have a parent (in case of the root node) or child nodes (in case of leaf nodes). Furthermore, to implement node partitioning, as described in Chapter 3, the tree information service also stores the identifiers of the set of physical nodes that implement a logical node. The service also stores the mapping table in the record to allow a physical node to determine which physical node in the set is responsible for storing the contact record for a specific object handle. Finally, to allow the record to be cached by a physical node, the record has a version number. Physical nodes use the version number to ensure that they use the newest version of the record. The size of the logical node record is almost entirely determined by the size of the mapping table since the size of the mapping table is in the order of 253 kilobytes (see Section 3.5.3) while the sizes of the identifiers and version number are between 4 and 8 bytes.

The physical node record stores the following properties of a physical node:

- The identifier of the physical node
- The identifier of the logical node it belongs to
- The network address
- The public key
- The geographical location
- A version number

The record starts with the identifier of the physical node that is described. The record also stores the identifier of the logical node the physical node belongs to, allowing a physical node to quickly determine the logical node it implements. The record also stores the contact information of the physical node, that is, its network address and public key. The physical node determines these two properties when it starts and stores them in the tree information service. The tree information service also needs to know the geographical locations of all the physical nodes that implement a logical node since it is responsible for (re)computing the mapping table when the partitioning of a node changes (see Section 7.3). The physical node record also has a version number to allow the record to be cached. The size of the physical node record is mostly determined by the size of the public key since this size is 128 bytes or 256 bytes (1,024 or 2,048 bits, see Chapter 6) while the sizes of the other values are between 4 to 8 bytes.

A physical node queries the tree information service when it is (re)started or when it expects or notices a change in the search tree structure. A physical node detects changes in the search tree by exchanging the version numbers of the logical and physical node record it uses as part of its normal communication. When the version number of the logical or physical node record used by a sending physical node is higher than the version number of the receiving node, the receiving node needs to retrieve the new record from the tree information service before it can proceed with the communication. When another node no longer responds to communication requests, the physical node also needs to check the tree information service to see whether the configuration information has changed. This indirect method of notifying nodes of changes is used only for nodes not directly involved in a change. Nodes that need to change or transfer location information are informed directly by the tree information service, as described in the next two sections.

The size of the configuration information stored by the tree information service is insignificant. The size of the information is formed by the sizes of the tables storing the logical and physical node records. To get an estimate of the amount of information to store, consider the example search tree from Chapter 3. This example search tree consisted of 2,430 logical nodes and 9,720 physical nodes. Thus, in this case, the tree information service needs to store 2,430 logical node records and 9,720 physical node records. Since the size of the logical node record is in the order of 253 kilobytes, the size of the table of logical node records is in the order of $2,430 \times 253$ kilobytes $= 615$ megabytes. If we assume the size of the physical node record is in the order of 160 bytes, the size of the table of physical node records is in the order of $9,720 \times 160$ bytes $= 1.6$ megabytes. The size of the configuration information is thus mainly determined by the table of logical node records. Note that since the example search tree represents only a minimal size that we want to support, the tree information service has to store more than 615 megabytes of configuration data.

The tree information service can be implemented by a single heavyweight server. Since the amount of data stored by the tree information service is not significant, this server can even store the configuration in main memory. However, to provide physical nodes with local access to the configuration information, we expect the tree information service to be implemented by a small group of heavyweight servers distributed across the network. Every server in the group keeps a replica of the complete configuration information. Since we are dealing with only a small group of servers, it is straightforward to keep the replicas (highly) consistent. Furthermore, since changes are likely to involve only a few physical or logical nodes, the size of the changes can be expected to be small.

To prevent frequent communication with the tree information service, physical nodes can cache the configuration information they received from the tree information service. By exchanging and comparing the version numbers of the logical and physical node records, physical nodes can easily determine whether they still use up-to-date configuration information.

7.3 Node Partitioning Changes

Recall from Chapter 3 that a logical node is implemented by a set of physical nodes. Contact records in the logical node are distributed across its physical nodes, with each physical node being assigned a subset of the contact records stored by the logical node as a whole. A physical node provides the actual storage and processing capacity needed by the subset of contact records assigned to it.

We described two methods to assign a contact record to a physical node in Chapter 3. The first method uses a hash function to distribute the contact records evenly over the physical nodes. The second method uses the geographical location field of the object handle to select the nearest physical node. The assignment of contact records to the physical nodes is determined by the mapping table of the logical node. This table is used by the physical nodes of other logical nodes to determine with which physical node to communicate.

The storage and processing capacity needed by a logical node can, however, change over time. For instance, when the domain associated with the logical node becomes more popular, more capacity is needed and a new physical node needs to be added. Also, the logical node might want or need to increase its processing capacity at a specific location in its domain, for instance, because a large number of objects reside at that particular location. When the set of physical nodes changes, the assignment of contact records to physical nodes changes as well, and the set of (existing) contact records of the logical node needs to be redistributed over the new set of physical nodes. The change in distribution of contact records depends strongly on the load-distribution scheme used by the logical node.

The changes to the partitioning of a logical node are thus about adding physical nodes

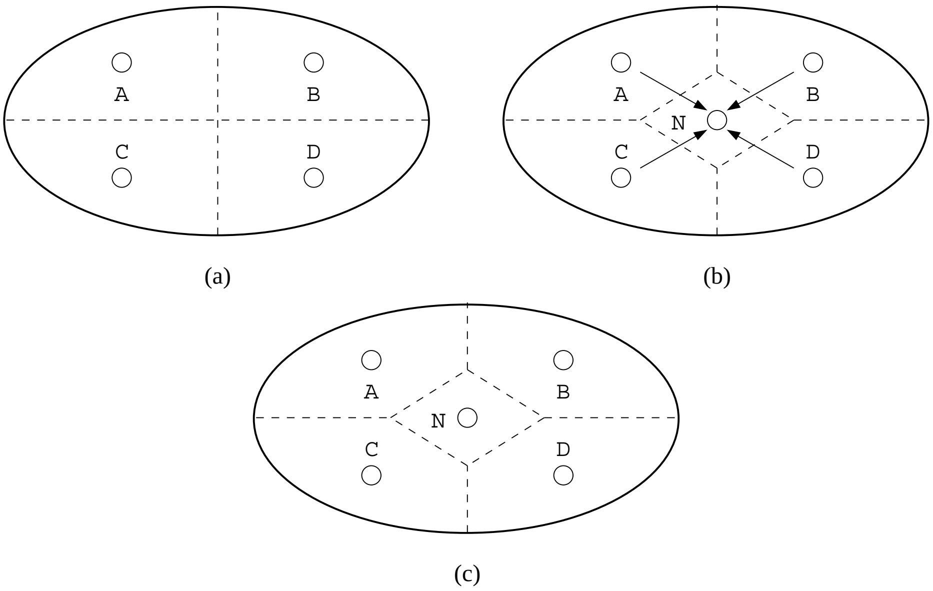


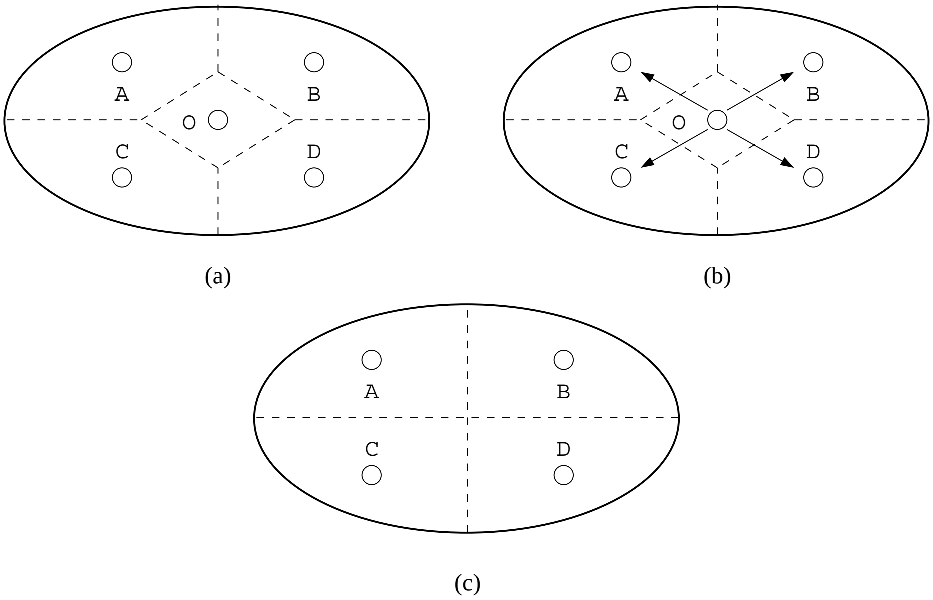
Figure 7.1: The transfer of contact records and change in mapping table caused by adding physical node *N* to the set of physical nodes implementing a logical node.

to and removing them from the set of physical nodes that make up a logical node. Using these two changes, we model the movement of a physical node inside a logical node (i.e., a change in network or geographical location) as the addition and the removal of a physical node. Note that replacing a physical node (i.e., replacing the machine running the node) does not require a partitioning change since the new machine simply takes over the role and identity of the old machine.

Figure 7.1 shows the transfer of contact records caused by adding a new physical node to a set of physical nodes. The figure also shows the associated domain of the physical nodes. The four physical nodes use the location-aware mapping scheme to distribute the set of contact records of the logical node they implement. In the figure the domain is partitioned in four parts to show which locations in the domain are mapped to which physical nodes. For instance, the contact records of all the object handles with a location in the upper-left quadrant are stored in physical node A.

Figure 7.1(a) shows the initial situation with the logical node implemented by four physical nodes A–D. In Figure 7.1(b) a new physical node N is added. As a result the domain is partitioned in five parts, and the contact records of the objects with a location closest to physical node N need to be transferred to node N. Figure 7.1(c) shows the final situation with the logical node implemented by five physical nodes, each having their own geographical area from where they support object handles.

Figure 7.2 shows the reverse situation where the transfer of contact records is caused



(a)

(b)

(c)

Figure 7.2: The transfer of contact records and change in mapping table caused by removing physical node *O* from the set of physical nodes implementing a logical node.

by the removal of physical node O. In this case all the contact records stored at physical node O need to be redistributed over the remaining four physical nodes A–D. Note that physical node O cannot disappear before all its contact records have been sent to the remaining physical nodes and their reception is acknowledged.

A new mapping table that reflects the new distribution of contact records over the new set of physical nodes has to be created for the logical node. This new mapping table needs to be distributed to other physical nodes that want to communicate with the logical node. We would like the redistribution of the contact records and the distribution of the new mapping table to be atomic. Unfortunately, redistributing the contact records may take a long time. For instance, even if a physical node that is to be removed is able to transfer its contact records with a speed of 10 megabytes per second, the transfer will still last 14 hours, given a contact record database of 500 gigabytes. We therefore have to deal specifically with a transition phase in which the transfer takes place and decide at which time during this phase the new mapping table is distributed to other physical nodes.

The main problem of this transition phase is dealing with lookup and update operations while the contact record transfer is in progress. During this phase, every contact record is either at its old physical node, waiting to be transferred, or at its new physical node, having just been transferred. This means that during the transition phase, other physical nodes wanting to send an update or lookup request have no way of knowing at which physical node a contact record is stored. Furthermore, the contact record might even be transferred while an update operation is in progress, introducing the risk of changes to a contact record being lost. A concurrency-control mechanism is thus needed to synchronize the transfer of contact records, the distribution of the mapping table, and the handling of lookup and update operations.

The main concept of our solution to dealing with this transition phase is that the physical node that ships out contact records remains responsible for those contact records during the transition phase. The physical node handles the transfer of contact records on a per contact record basis: Either the physical node still stores the contact record and can handle update and lookup operations itself, or it can forward the update or lookup request to the physical node that currently stores the contact record. This way the physical node acts as a proxy server during the transfer of contact records and the mapping table used before the transition phase can remain in effect during the transition phase. This ensures that the transition phase is transparent to other physical nodes that want to communicate with the logical node during the transition phase.

Consider the simplified situation of a single change in the set of physical nodes (i.e., one physical node being added or removed). We distinguish three steps during the transition phase:

1. **Initialization** step. The tree information service is informed of the new situation and requested to compute a new mapping table. This mapping table is kept hidden, in the sense that it is distributed only to the physical nodes that need to transfer contact records.

2. **Transfer** step. The contact records are sent to their new locations using the new, hidden mapping table. During this step, the physical nodes of other logical nodes still use the old mapping table. However, the physical nodes transferring contact records use the new, hidden mapping table to forward requests to the physical node that stores the requested contact record.

3. **Finalization** step. All the physical nodes sending contact records inform the tree information service when their transfer is done. When the tree information service has been informed by all physical nodes, the service makes the new mapping table official and distributes it to all physical nodes that want to communicate with the logical node.

For example, in Figure 7.2 where physical node `O` is removed, node `O` sends its contact records to physical nodes `A`–`D`. During this transfer node `O` remains responsible for the contact records previously stored by it. It will either handle the lookup and update request itself or forward the request to the record's new location. Node `O` uses the new, hidden mapping to decide to which of the physical nodes `A`–`D` it forwards update and lookup requests.

In the general case, we need to deal with concurrent changes, that is, new changes that are made while the logical node is still in the transition phase of a previous change. This case is especially likely in larger domains, such as the root domain. We can, in principle, use the same method as described above to deal with this case. Every time a new change is sent to the tree information service, a new mapping table is computed, possibly using the previous hidden mapping table. All physical nodes involved in the new change are informed and start their transfer. However, the physical nodes still involved in the old transfer ignore the new change until they are ready; only then do they consider the new change.

Several issues have to be worked out for the general case. For instance, methods have to be developed to determine how a physical node knows where to forward requests to, when the transition phase has completed, and when the new hidden mapping table can be made official. Also, if we want to support the hashing load-distribution scheme, we need to deal with the redistribution of contact records while using this scheme. These issues are left unresolved in this dissertation.

7.4 Search Tree Changes

In this section we describe (some of) the issues relating to changes to the logical structure of the search tree. Recall from Chapter 2 that the logical structure of the search tree is a representation of the domain hierarchy, and that the domain hierarchy, in turn, is a representation of the distances in the underlying network. For example, clients that share a leaf domain are closer to each other than clients that share only the root domain. The metric used by the location service for network distances (and thus locality) is derived from a combination of geographical distance, different network metrics (e.g., latency and hop count), and the structure of the organizations using the network. To ensure the locality of lookup and update operations, the search tree and thus the domain hierarchy need to be good representations of distances in the network.

There are two main causes for changes to the logical structure of the search tree. The first cause of change is the initial deployment of the location service. During this phase the search tree will grow and cover new parts of the underlying network as clients want to insert contact addresses at new locations. Also during this phase, parts of the network already covered by an existing domain will be (also) covered by new, increasingly smaller child domains. The introduction of child domains provides a more detailed representation of distances in those domains.

The second cause of change is the need to adapt the search tree to significantly changed distances in the network. This happens mainly after the initial deployment of the location service. These changes in distance are caused by changes in the underlying network or the organizations using the network. We expect the logical structure of the search tree to change only infrequently after the deployment since significant changes in the underlying network or organizations are rare.

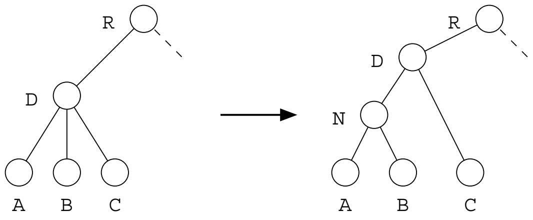


Figure 7.3: Example 1: Search tree changed caused by an improvement of the local network.

7.4.1 Example Changes

To better understand the causes of change and their resulting changes, we consider three example situations where a change in the environment of the location service results in a change in the logical structure of the search tree.

Example 1: Adding a New Network Connection

The first example is a network change where some distances in the network are shortened by the addition of a high-speed connection. This situation is shown in Figure 7.3. In the initial situation the network is covered by intermediate node D. The domain of node D is divided into three leaf domains represented by leaf nodes A, B, and C. Now assume that a new high-speed connection is added between the leaf domains of leaf nodes A and B, shortening the network distance between them but leaving them far enough apart to warrant distinct leaf domains.

In this case we need to add a new intermediate domain with a new intermediate node N to the search tree. Node N represents the fact that the domains of leaf nodes A and B are closer to each other than to the leaf domain of node C. These three leaf domains are, however, still part of the existing domain of node D.

Example 2: Changing ISP

The second example is also a network change. In this case, however, an organization decides to change the ISP it uses to connect its LAN to the Internet. This situation is shown in Figure 7.4. In the initial situation the LAN is covered by the leaf domain of leaf node C. Since the leaf domain is part of the domain of the ISP, represented by intermediate node F, leaf node C is a child node of node F. When the organization starts using the new ISP, represented by intermediate node G, its LAN becomes part of the domain of node G. Leaf node C therefore needs to become a child node of node G instead of node F.

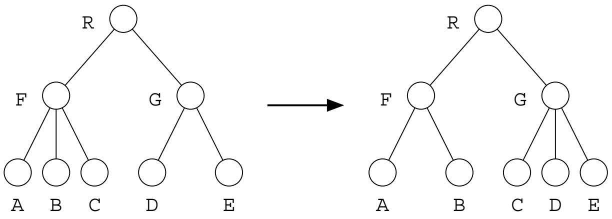


Figure 7.4: Example 2: Search tree changed caused by a change of ISP.

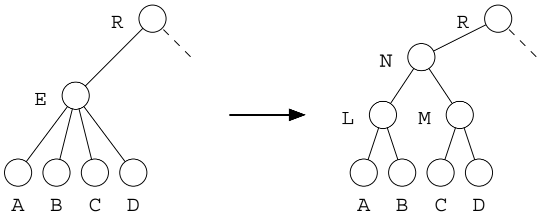


Figure 7.5: Example 3: Search tree changed caused by an organizational split.

Example 3: Splitting a Company

The final example is an organizational change where a unified company splits into two divisions. This situation is shown in Figure 7.5. In the initial situation the network of the company is covered by the domain of the intermediate node `E` and the leaf domains of the leaf nodes `A`–`D`. Now assume that the two new divisions split the network in two, with one division using the network covered by the domains of leaf nodes `A` and `B` and the other using the network covered by the domains of leaf nodes `C` and `D`.

To model the new situation in the location service, we need to introduce two new intermediate domains with intermediate nodes `L` and `M`. The new domains cover the networks of the two divisions. If we assume that the divisions still work closely together, we also want to introduce a third domain covering both divisions. This domain is represented by intermediate node `N`.

7.4.2 Basic and Composite Changes

When transforming the structure of the search tree, we face the problem of what to do with the existing location information (i.e., contact addresses and forwarding pointers) stored in the search tree. After we have changed the search tree, the information still needs to fulfill the consistency requirements described in Chapter 2. The connectivity requirement C2 is particularly important. It states that “For each node N, the contact record for object O at node N stores a forwarding pointer to a child node of N if and only if the contact record for O at that child is not empty.”

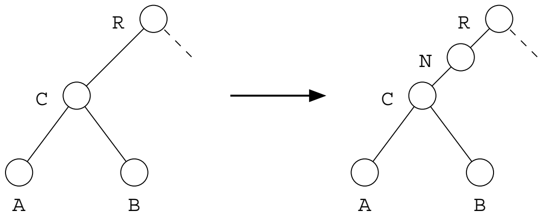


Figure 7.6: Basic change `insert-node`: Inserting an intermediate node.

To ensure that this consistency requirement is met in the modified search tree, the location information needs to be updated. For instance, contact records need to be created at new logical nodes, contact records need to be modified at existing logical nodes, forwarding pointers need to be inserted, and contact address need to be transferred. Changes to the logical structure of the search tree are therefore more complex than changes to the partitioning of a logical node (described in the previous section) where contact records only needed to be transferred.

When making changes to the search tree, we are faced with an atomicity problem similar to the one described in the previous section. When we want to make a change to the search tree, we need to update two kinds of information: the configuration information in the tree information service and the location information stored in the search tree. Unfortunately, updating the location information in the search tree might take a long time. We therefore need to deal with a transition phase in which the search tree changes its structure and concurrent update and lookup operations can still use the location information in the search tree.

To simplify this problem, we divide the complex changes to the structure of the search tree into four basic changes: `insert-node`, `remove-node`, `split-node`, and `merge-nodes`. These basic changes are simple enough to guarantee correct results from concurrent operations, but still allow us to perform complex changes when combined into composite changes. To allow us to grow the search tree during the initial deployment of the location service, we add a fifth operation that appends a new leaf node to an existing leaf node: `append-leaf`.

Figures 7.6–7.9 show examples of these basic changes. Figure 7.6 shows a new intermediate node `N` being inserted in between the root node `R` and intermediate node `C`. Figure 7.7 shows an old intermediate node `O` being removed, resulting in a direct connection between root node `R` and leaf nodes `A` and `B`. Figure 7.8 shows two inverse operations: splitting and merging nodes. Node `E` is split into two nodes `N` and `M`, or nodes `N` and `M` are merged into a single node `E`. Figure 7.9 shows a new leaf node `N` being appended to the leaf node `A`, making leaf node `A` an intermediate node.

The `insert-node` and `append-leaf` basic change result in an increase in the height of the search tree. This increase in the height affects, unfortunately, the efficiency of the lookup and update operation adversely. Specifically, operations that need to visit

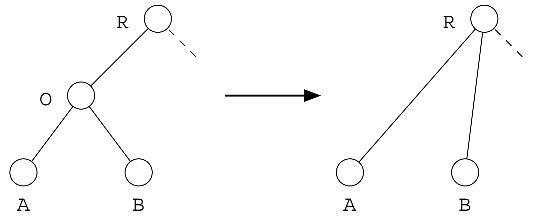


Figure 7.7: Basic change `remove-node`: Removing an intermediate node.

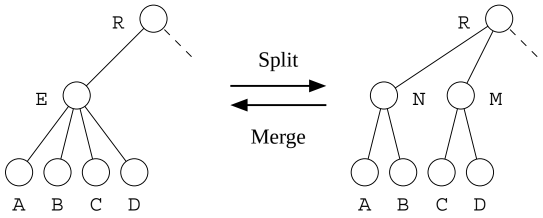


Figure 7.8: Basic changes `split-node` and `merge-nodes`: Splitting and merging nodes, respectively.

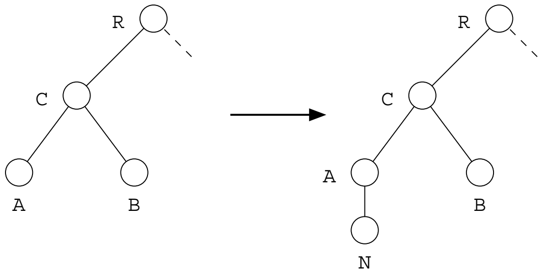


Figure 7.9: Basic change `append-leaf`: Appending a leaf node.

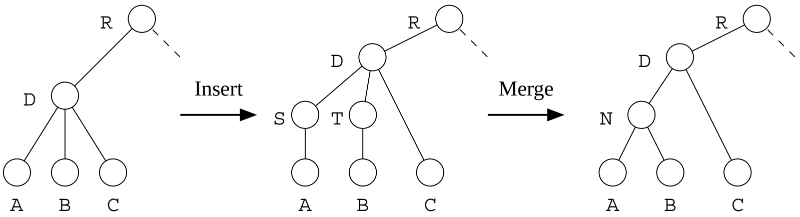


Figure 7.10: Implementation of the search tree change of Example 1 using the `insert-node` and `merge-nodes` basic change.

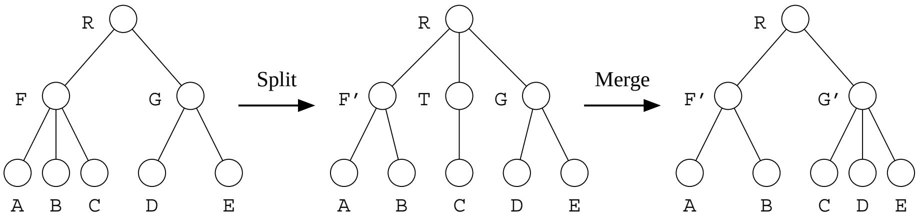


Figure 7.11: Implementation of the search tree change of Example 2 using the `split-node` and `merge-nodes` basic change.

the root node take longer to complete since they need to visit more nodes to reach the root node in the search tree. Examples of such operations are an insert operation that inserts the first contact address of an object and a lookup operation on an object with a single replica located far away. The management organization therefore needs to take care not to indiscriminately insert and append new logical nodes to the search tree.

Figure 7.10 shows how the complex change of Example 1 can be implemented using the `insert-node` and `merge-nodes` basic change. In the first step, we insert two temporary intermediate nodes `S` and `T` between node `D` and node `A` and node `B`, respectively. In the second step, we merge these two temporary nodes into a single intermediate node `N` that covers the leaf domains of leaf nodes `A` and `B`.

Figure 7.11 shows how the complex change of Example 2 can be implemented using the `split-node` and `merge-nodes` change. The leaf node can be moved to its new position in the search tree by first splitting intermediate node `F` into node `F′` and temporary node `T`, and then merging temporary node `T` and node `G` to form the new intermediate node `G′`.

Figure 7.12 shows how the complex change of Example 3 can be implemented using the `insert-node` and `split-node` changes. In the first step, we insert a new intermediate node `N` between node `E` and its parent node `R`. In the second step, we split the existing intermediate node `E` into two new intermediate nodes `L` and `M`. We make leaf nodes `A` and `B` a child of node `L` and leaf nodes `C` and `D` a child of node `M`.

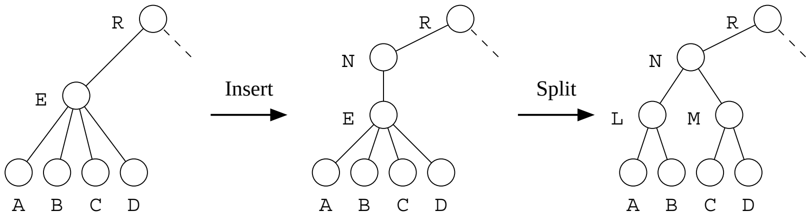


Figure 7.12: Implementation of the search tree change of Example 1 using the `insert-node` and `split-node` basic change.

7.4.3 Implementing Search Tree Changes

To implement our five basic changes, we still need to deal with their respective transition phases. To deal with the transition phase of each basic change, we propose a solution similar to the one presented in the previous section. The solution consists of two components. First, the changes to the contact records in a node are made on a per contact record basis. Second, the nodes in the search tree involved in the change behave in a proxy-like fashion, that is, they either perform a requested operation themselves or forward it to the correct logical tree node. The exact method to allow the maximum amount of concurrency possible is still part of ongoing research.

`insert-node`**: Inserting a Logical Node**

Figure 7.13 shows the process of inserting a new node `N` between parent node `P` and child node `C`. To perform this change, we need to inform parent node `P` that node `N` has taken the place of its (former) child node `C`, and we need to inform child node `C` that node `N` has taken the place of its (former) parent node `P`. Furthermore, we need to restore the connectivity of the paths of forwarding pointers running through node `C` (and thus node `N`). We restore the connectivity during the transition phase by inserting forwarding pointers at the new node `N`. Adding the forwarding pointers reconnects the paths of forwarding pointers flowing through nodes `P`, `N`, and `C`, and thereby allows contact addresses in the domain associated with node `N` to be found. At parent node `P`, we need to redirect the forwarding pointers from node `P`'s former child node `C` to its new child node `N`.

To store forwarding pointers at the new node `N`, we need to know which object handles have a contact record stored at intermediate node `C`. Node `C` therefore goes through its contact record database and requests node `N` to insert a forwarding pointer for every contact record found. To redirect the forwarding pointers at node `P`, node `P` needs to go through its database and change all forwarding pointers pointing to child node `C` to point to the new node `N`. Note that redirecting these pointers might actually consist of simply changing a single logical node identifier shared by all forwarding pointers.

To deal with the transition phase, as shown in Figure 7.13(b), we need to temporarily change the behavior of the new node `N`. The transition phase starts when nodes `P` and

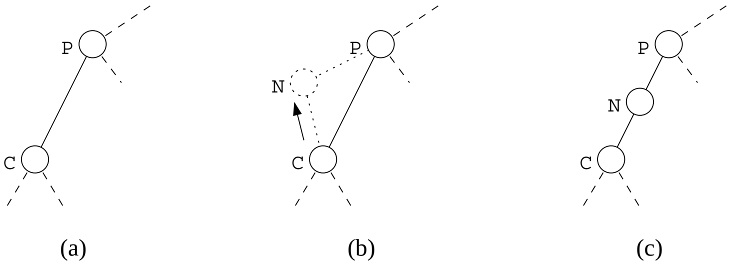


Figure 7.13: Basic change `insert-node`: (a) Initial situation. (b) In the transition phase node `C` requests the insertion of forwarding pointers at node `N`. (c) Final situation.

`C` start using node `N`. During the transition phase node `N` will be increasingly filled with contact records, which store a single forwarding pointer to node `C`. Since nodes `P` and `C` forward requests to node `N` and node `N` is not yet completely filled, node `N` receives requests for both known and unknown objects.

For known objects, node `N` has a contact record, and it can simply handle all requests for those objects. Node `N` can also handle insert requests for unknown objects since these will simply insert a contact address or forwarding pointer, and request confirmation from parent node `P`. Lookup and delete requests for unknown objects, however, require special actions. Node `N` needs to forward these requests to the proper node, that is, to its parent node `P` if the request came from its child node `C`, and vice versa. The transition phase is finished when node `C` has checked all its contact records.

`remove-node`: Removing a Logical Node

Figure 7.14 shows the process of removing an existing node `O` from parent node `P` and child nodes $\mathtt{C}_1$ and $\mathtt{C}_2$. To perform this change, we need to inform parent node `P` that child nodes $\mathtt{C}_1$ and $\mathtt{C}_2$ have taken the place of its (former) child node `O`, and we need to inform child nodes $\mathtt{C}_1$ and $\mathtt{C}_2$ that node `P` has taken the place of their (former) parent node `O`.

When we want to remove an existing node, we first need to transfer the location information (i.e., contact addresses and forwarding pointers) it stores to its parent node `P`. This transfer of location information takes place during the transition phase. Recall that the location information is stored in the contact records of node `O`, and that these contact records consist of a set of contact fields, one field for each child node $\mathtt{C}_n$. In Figure 7.14 the contact records consists of two contact fields. These contact fields will replace the single contact field for node `O` in the contact records at parent node `P` since parent node `P` will become the parent of the child nodes.

To move the contact records to parent node `P`, node `O` goes through its contact record database and transfers its contact records to node `P`. Node `P` retrieves the related contact record (i.e., associated with the same object handle) from its contact record database and

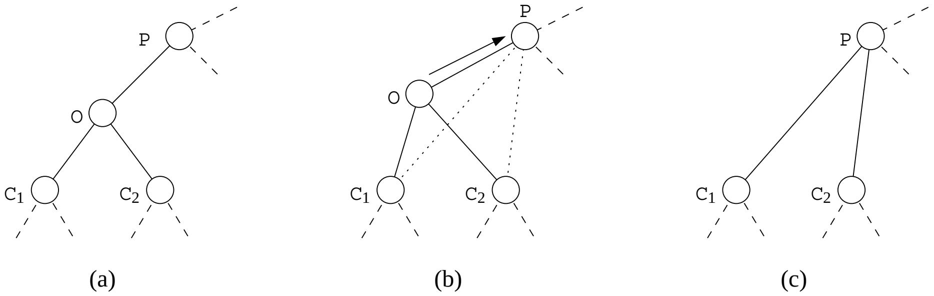
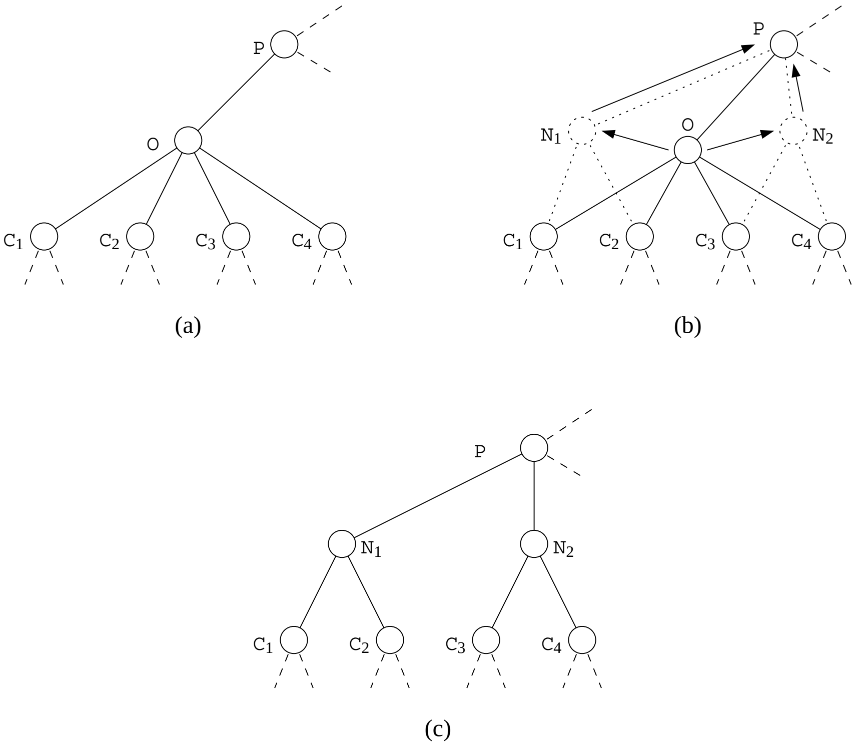


Figure 7.14: Basic change `remove-node`: (a) Initial situation. (b) In the transition phase node `O` transfers its contact records to parent node `P`. (c) Final situation.

removes the existing contact field for node `O` from the record. It then merges the remaining contact fields of the contact record with the contact record received from node `O`. Note that during the transition phase, parent node `P` stores two types of contact record, that is, contact records with a contact field for node `O` and contact records with contact fields for nodes $\mathtt{C}_1$ and $\mathtt{C}_2$. Node `O` removes the contact record when it has been received by node `P`.

To deal with the transition phase, nodes `O` and `P` have to change their behavior slightly. Node `P` uses the contents of the contact record to determine its behavior. If the contact record has a forwarding pointer to node `O` (i.e., no transfer has taken place for this contact record), node `P` sends its request to node `O`; if it has a forwarding to a child node $\mathtt{C}_n$, it sends its request to that child node. Node `O` behaves in the proxy-like fashion, as indicated before. It handles requests for known objects in the normal fashion since it still stores their contact records. Requests for unknown objects from its children are forwarded to its parent node `R`. Requests for unknown objects from parent node `P` (i.e., lookup requests) can simply be ignored since the path of forwarding pointers followed by the lookup operation is apparently being deleted. Child nodes $\mathtt{C}_1$ and $\mathtt{C}_2$ simply keep on using node `O`, unaware of the transition phase. The transition phase is ready when node `O` has transferred all its contact records to node `P`. At this time we inform child nodes $\mathtt{C}_1$ and $\mathtt{C}_2$ that node `P` is their parent, and node `O` can be deleted.

There is one situation we have to deal with separately. This situation occurs when a contact record at parent node `P` stores a contact address in its contact field for node `O`. In this situation node `P` needs to determine in which of the new contact fields the contact address has to be stored. In our example, the node chooses between the contact field of node $\mathtt{C}_1$ and the contact field of node $\mathtt{C}_2$. Node `P` can determine the contact field using the identifier of the leaf node where the contact address was originally inserted. Recall from Chapter 4 that this identifier is part of the contact address. We know node `O` is empty in this case since it follows from our consistency rule C2 that no contact addresses or forwarding pointers can be stored in node `O`.



(a)

(b)

(c)

Figure 7.15: Basic change split-node: (a) Initial situation. (b) In the transition phase existing node O transfers its contact fields to the new nodes N_1 and node N_2. The new nodes, in turn, insert forwarding pointers at parent node P. (c) Final situation.

split-node: Splitting a Logical Node

Figure 7.15 shows the process of splitting an existing node O into two new nodes N_1 and N_2. Splitting a node requires different actions depending upon whether we split a leaf node or an intermediate node. When we split an intermediate node, as shown in Figure 7.15, we assign a subset of child nodes of node O to node N_1 and the rest to node N_2. In the example, nodes C_1 and C_2 are assigned to node N_1 and nodes C_3 and C_4 are assigned to node N_2.

As a consequence of the split, the location information stored by node O needs to be redistributed over nodes N_1 and N_2. In practice, this means that the contact fields of the child nodes assigned to node N_1 need to be transferred from node O to node N_1 and that the rest of the contact fields need to be transferred to node N_2. Furthermore, the contact fields for node O at parent node P need to be split in two to represent the two new nodes

N_1 and N_2. The two new contact fields in the contact records of node P need to store a forwarding pointer if node N_1 or node N_2 stores a contact record, respectively.

Splitting the domain associated with a leaf node requires a different approach. When we split a leaf domain into two domains, we need a way to determine how to distribute the location information (i.e., contact addresses) of the original leaf domain over the two new leaf domains. Since leaf domains are atomic, the contact records in a leaf node consist of only a single contact field to store contact addresses for objects. We therefore cannot distribute the location information based on the original contact field it comes from. In our ongoing research, we are considering two approaches to determine how to distribute the location information.

In the first approach we divide the world up into small atomic cells with a fixed size and location, based on network distances, geography, and administrative boundaries. These cells are significantly smaller than leaf domains, and every leaf domain is assigned a set of adjacent cells. Every contact address stores, besides its leaf domain, also the cell identifier of the cell to which it resides. When a leaf node splits, the two leaf nodes cover distinct sets of adjacent cells, and the contact addresses can be distributed based on the cells they belong to. The downside to this approach is that is not obvious how to divide the network into cells with a fixed location that is useful and that you have to get the division into cells right the first time.

The second approach uses the contact address lease system, discussed in Chapter 2. The lease system forces clients to regularly contact the location service to extend their lease. At the renewal time, the client will notice the old leaf node is split. The client can then delete its contact address from the old leaf node and reinsert the address using the new leaf domain structure. Only when the lease of the last contact addresses has expired is the transfer completed, and can the old leaf node disappear. The downside of this approach is that it may take a long time to complete the transfer. Note that with this approach the leaf nodes are not involved in the actual transfer of contact addresses.

The process of distributing the contact fields of a node (leaf or intermediate), as shown in Figure 7.15(b), consists of three steps. In the first step, parent node P adds empty contact fields for nodes N_1 and N_2 to all its contact records, allowing forwarding pointers to be inserted. In the second step, the contact fields of node O are divided between nodes N_1 and N_2. Node O removes the contact record when it is received by nodes N_1 and N_2. For every set of contact fields that node N_1 (or N_2) receives, it contacts parent node P to insert a forwarding pointer in the new contact field for node N_1 (or N_2) at node P. In the third step, node P removes all the old contact fields for node O.

During the transition phase node O behaves like a proxy. It forwards requests to nodes N_1 and N_2 for objects that are unknown and handles the operations itself for objects it does know. The behavior of node P depends upon the contents of its contact record. If one of the contact fields for nodes N_1 and N_2 is filled, the contact field for node O is ignored. Otherwise, the contact field of node O is used. Nodes N_1 and N_2 handle all requests normally. They do send their requests directly to child nodes C_1–C_4 (i.e., not via the old node O). The transition phase is finished when all the contact fields from node O have been transfered to nodes N_1 and N_2. At this time node O can be deleted.

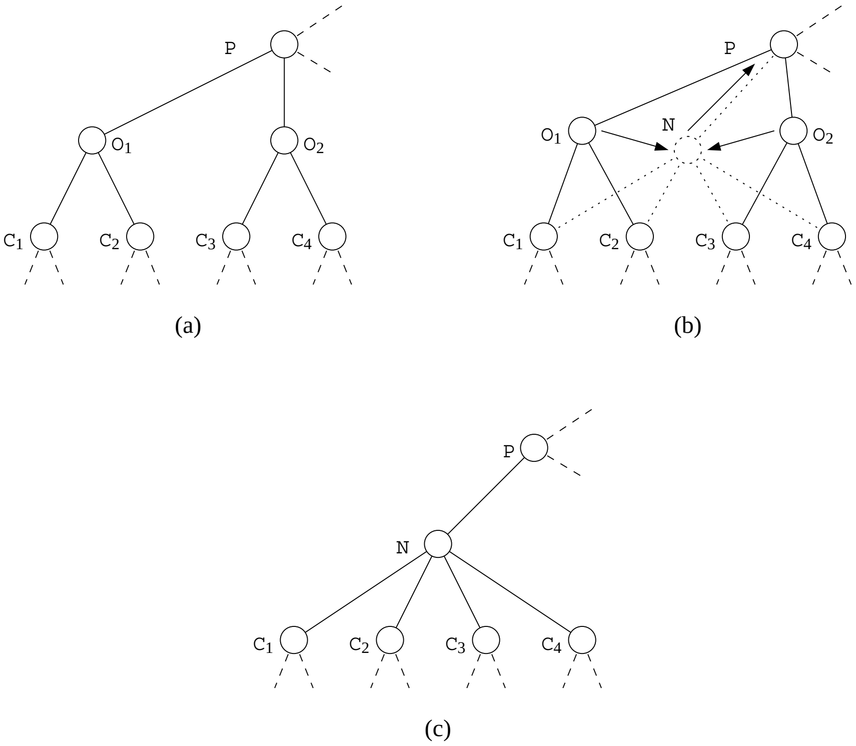


Figure 7.16: Basic change `merge-nodes`: (a) Initial situation. (b) In the transition phase existing nodes $\mathtt{O}_1$ and node $\mathtt{O}_2$ transfers their contact fields to the new node `N`. The new node, in turn, inserts forwarding pointers at parent node `P`. (c) Final situation.

There is one situation we have to deal with separately. This situation occurs when a contact record at parent node `P` stores a contact address in its contact field for node `O`. In this case we have to determine in which of the new contact fields to store the contact address. This is the same situation as encountered during the `remove-node` basic change. In this case we can also use the leaf node identifier to determine the correct contact field to store the contact address.

`merge-nodes`: Merging Two Logical Nodes

Figure 7.16 shows the process of merging two existing nodes $\mathtt{O}_1$ and $\mathtt{O}_2$ into a single new node `N`. When two nodes are merged into one node, we also need to merge the contact records of the two nodes. As a consequence, we also merge the contact fields of nodes $\mathtt{O}_1$ and $\mathtt{O}_2$ into a single contact field at parent node `P`.

This process of merging two nodes, as shown in Figure 7.16(b), consists of three steps. In the first step, node P adds a contact field for node N to every contact record it stores, allowing forwarding pointers for node N to be inserted. In the second step, nodes O_1 and O_2 send their contact records to node N which merges them into a single contact record. Nodes O_1 and O_2 remove their contact records when it is received by node N. Node N requests a forwarding pointer at node P for every newly created contact record. In the third step, node P removes its contact fields for nodes O_1 and O_2.

During the transition phase nodes O_1 and O_2 behave like proxies. They forward requests to node N for objects that are unknown, and handle the operations themselves for objects they do know. The behavior of node P depends, as when splitting a node, upon the contents of its contact record. If the contact field for node N is filled, the contact fields for nodes O_1 and O_2 are ignored. Otherwise, the contact field of nodes O_1 and O_2 are used. Node N handles all requests normally. It does send its requests directly to child nodes C_1–C_4 (i.e., not via the old nodes O_1 and O_2). The transition phase is finished when all the contact records from nodes O_1 and O_2 have been transfered to node N. At this time nodes O_1 and O_2 can be deleted.

There is one situation we have to deal with separately. This is the case that node P stores a contact address in its contact field for either node O_1 or O_2. This might present a problem for consistency rule R3 which states: "A contact field can contain either a forwarding pointer or a set of contact addresses, but not both." If the contact field for node O_1 stores a forwarding pointer and the contact field for node O_2 stores a contact address, merging these two fields will lead to a violation of consistency rule R3. The avoid this violation, the contact address needs to be moved down to node N. This transfer increases, unfortunately, the complexity of this basic change.

append-leaf: **Appending a Logical Leaf Node**

When a new leaf node N is appended to an existing leaf node L, the existing node L becomes an intermediate node, as shown in Figure 7.17. To ensure a consistent search tree, we need to update two aspects of the location information stored. First, since leaf nodes are the default storage location for contact addresses, we need to move the contact addresses stored at the intermediate node L to the new leaf node N. Second, recall that to support the move-down operation in Chapter 4, we store the identifier of the leaf node in the contact address. Since the contact address will now be stored at a new leaf node, this identifier has to be updated in all contact addresses to reflect the new leaf node N.

The operation to transfer a contact address to the new leaf node is similar to the move-down operation described in Chapter 4 since it consists of inserting the contact address at the new leaf node N and inserting a forwarding pointer at the existing node L. We can, conceivably, even reuse the move-down operation by changing the identifier in the contact addresses at the intermediate node L to the identifier of the new leaf node N. We can then simply wait until the intermediate node L notices its addresses are stable and invokes the move-down operation at the new leaf node L. Using this method we do not need to deal explicitly with the transition phase since move-down operations are part of the normal

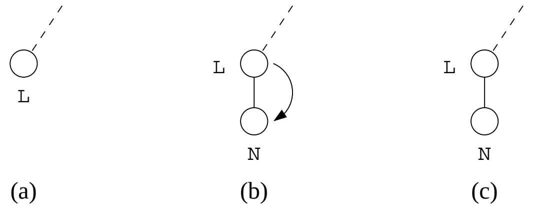


Figure 7.17: Basic change `append-leaf`: (a) Initial situation. (b) In the transition phase the move-down operation transfers contact addresses to the new leaf node. (c) Final situation.

operations of the nodes anyway.

7.4.4 Open Issues

In this section, we have side-stepped several issues that require future research to come to a complete and efficient method of dealing with changes to the logical structure of the search tree. In this final part of the section, we address some of these issues briefly.

In our description of changes to the search tree, we discussed all operations in terms of logical tree nodes. However, in practice most of these logical nodes will be implemented using multiple physical nodes. Since our operations work on a per contact record basis, the node partitioning does not threaten our solution. All physical nodes simply implement the changes described for the logical node. It does, however, add an extra layer of complexity. A specific problem that has to be addressed is the interaction of node partitioning changes and search tree changes (e.g., adding physical nodes to a logical node that is being inserted). In the worst case, we might have to allow nodes to be involved in only one type of change (partitioning or search tree) at a time.

The method described in this section was influenced by our desire to first develop simple changes that we could oversee, and only in a second step add complexity by combining them. While this method leads to a conceptually simple model, it does not result in efficient complex changes. A third step is therefore required to optimize the frequently occurring complex changes to the search tree. A second efficiency problem is that our method requires the introduction of new independent logical nodes, for instance, during a node split. This can, unfortunately, lead to problems since it requires serious amounts of resources when splitting a node located high in the tree. Intuitively, it should be possible to split a logical node by splitting and reusing the set of physical nodes that implement the logical node. Such a method does, however, require further research.

The choice of basic changes to support was made more or less intuitively. To ensure the completeness of our solution to dealing with changes to the search tree, however, we need to do research on whether we can actually perform all relevant complex changes using our basic changes.

Chapter 8

Prototype Implementation

This chapter discusses a prototype implementation of the Globe location service, and focuses specifically on the measurements made of this prototype. These measurements assess the performance of the location service, and are intended to show that the performance of individual physical nodes does not hinder the scalability of the location service as a whole. To measure the performance of physical nodes under different types of conditions, measurements were performed with different configurations of the search tree. The main results from the measurements are that a physical node can support at least 9 update operations or 107 lookup operations per second. From these results we conclude that a scalable location service can be built.

8.1 Building a Location Service Prototype

In the previous chapters, we have examined the behavior of the location service by analyzing and simulating the interaction between its nodes. In this chapter, however, we focus on the performance of an actual implementation. Since the location service consists mostly of the physical nodes that constitute its search tree, the performance of the location service is strongly influenced by these physical nodes. It is therefore important that these physical nodes perform efficiently since that enables the location service to be scalable. To examine whether an efficient location service can be implemented, we implemented a prototype and measured its performance, focusing on the performance of the physical nodes.

The process of implementing the prototype, however, also had additional advantages. For instance, it allowed us to verify that the functionality of the location service was complete. It also showed us that the we could translate the update and lookup algorithms from our high-level Ada-like pseudocode, as presented in the previous chapters, into executable code. The prototype of the location service is also used in a Globe system that runs continuously. This system is used to demonstrate the functionality of Globe, and serves as a testbed for performing experiments for subprojects of Globe, such as the Globe Distribution Network (GDN) [Bakker, 2002] and GlobeDoc [Kuz, 2003].

8.2 Assessing the Performance of the Prototype

The performance of the location service can be characterized using three views. The first view is the **user view**. This view deals with the question of how long an update or lookup operation takes in the location service, and is of interest to the users of the location service. For the location service to be usable, a user should not have to wait a long time for an operation to be finished, that is, the **duration** of update and lookup operations should be short.

The second view is the **system view**. This view deals with the question of how many update or lookup operations per second the location service can handle, and is of interest to the operators of the location service. We call this value the **throughput** of the location service. For the location service to be maintainable, the number of physical nodes that implement the location service should be as low as possible. To achieve a low number, the (maximum) throughput of individual physical nodes has to be high.

To explain the results from the user and system view, we also looked inside the physical nodes. This third view is called the **engineering view**. It examines the distribution of the total time spent in a physical node over the various software modules that implement the node. This view allows us to see where further research is needed to improve the performance of physical nodes.

8.3 Prototype Implementation

The prototype implementation of the Globe location service can be divided into three parts: the search tree, the tree information service, and the resolver library. The main part of the prototype is the search tree, which consists of a set of physical nodes. The implementation of the physical node is the main subject of this section. The tree information service is a simple process that stores the structure of the search tree and the contact information of the physical nodes, and which allows the physical nodes to query this information (see also Chapter 7). The resolver library is linked with client programs, and allows these programs to easily invoke the insert, delete, and lookup operations in the location service. This library basically consists of the code needed to communicate with the nearest leaf node of the search tree.

As shown in Figure 8.1, the design of the physical node roughly follows the design described in the previous chapters. This design consists of four layers, with the top layer divided into three parts. The top layer implements the insert, delete, and lookup algorithms. To improve the engineering view of this layer, we separated the location cache and contact record database (CRDB) modules from the rest of this layer. The RPC layer contains the implementation of the RPC mechanism (i.e., the sending and receiving of request and reply messages), including the persistent message log used for crash recovery.

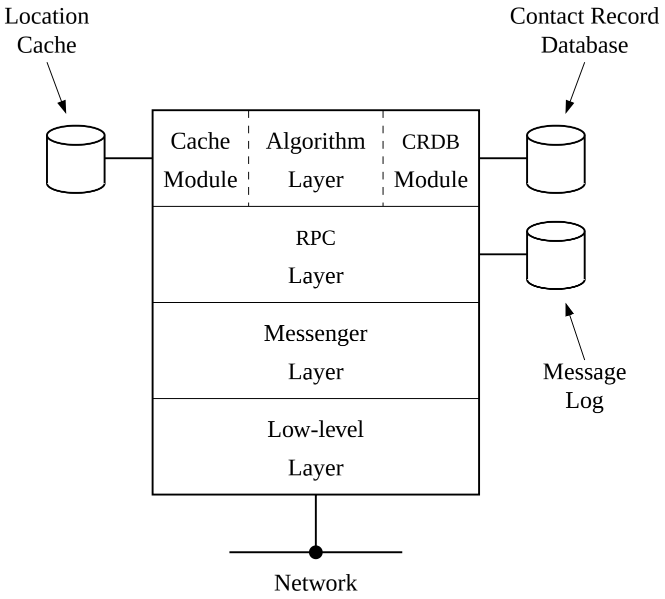


Figure 8.1: Global design of the physical node prototype.

This layer also implements the scheduling of procedures and the distribution of load over multiple physical nodes. The messenger layer implements reliable, ordered communication between two physical nodes, including the marshalling of network packets. Finally, the low-level layer implements the low-level support for networking (e.g., socket calls), threading (e.g., locking), and timers.

The design of the physical node is driven by the use of pop-up threads. This means that there are threads waiting in the low-level layer for timer or network events. Once such an event occurs (e.g., the arrival of a packet), a thread will “pop up” from the low-level layer, and handle the event by calling the appropriate callback function in a higher-level layer, usually the messenger layer. As an indirect result of the event, one of the procedures in the algorithm layer can be invoked.

The design is mostly single threaded, that is, at any point in time, only a single thread can be active in the top three layers. The low-level layer, however, is not single threaded, but contains two concurrent threads: one thread waiting for network events and one thread waiting for timer events. The low-level layer uses a locking mechanism to ensure that at any time at most one of these two threads is executing a callback function in the higher-level layers. The use of a single thread of control greatly simplified the design of the physical node since the need for concurrency control was limited to the low-level layer. Using a single-threaded design can, however, limit the performance of a physical node, especially on multi-processor hosts.

An important drawback of using a single-threaded design is that a pop-up thread is not allowed to block in the higher-level layers. Otherwise, the physical node would become idle since both pop-up threads would not be able to handle new events. The blocked thread (e.g., the network thread) cannot accept new events because it is blocked; the other thread (e.g., the timer thread) cannot handle new events since the thread is not allowed to enter the (single threaded) higher-level layers. The pop-up threads therefore have to run to completion, and return to the low-level layer.

Unfortunately, the update and lookup procedures in the algorithm layer do need to block in a higher-level layer when performing an RPC to allow other procedures to start or continue executions, as described in Chapter 5. We solved the blocking-thread problem by using continuations [Milne and Strachey, 1976]. A **continuation** is a piece of data that allows a previously interrupted procedure to continue its execution. In the physical node, the update and lookup procedures save their state in a continuation before starting an RPC, and the RPC layer uses the continuation to continue the procedures when the RPC has completed. When a procedure has saved its state and send its RPC request, its thread can return to the low-level layer to accept new work.

The current prototype does not implement all the functionality described in the previous chapters. The three most important parts that are missing are the move-up and move-down operations for contact addresses (described in Chapter 4), the stable storage for the contact record database and message log (described in Chapter 5), and the security features (described in Chapter 6). Since tree management (described in Chapter 7) is not yet fully developed, it is also not included in the prototype, which significantly simplified the design of the tree information service. The performance of the prototype is also limited because we used only a straightforward design, and did not (yet) try to optimize its design for high performance.

For the measurements of the engineering view, we added timers to various modules and layers of the physical node prototype. These timers measured the total amount of time spent in the various modules and layers. Care had to be given to the timing of the low-level layer since that layer contained two concurrent threads and the time spent there should not be measured twice.

We implemented the prototype using the Java programming language. Specifically, we used the Java 2 SDK 1.3.1 from Sun. This Java distribution includes a just-in-time (JIT) compiler to translate Java bytecode into native code. We used this JIT compiler in all our measurements. To simplify the implementation of persistent storage, for instance, for the contact record database, we used the Berkeley database library (version 3.3.11). We used the standard Java remote method invocation (RMI) protocol and libraries to implement the communication between the physical nodes and the tree information service. The size of the complete prototype was around 50,000 lines of code.

8.4 Overall Measurement Methodology

In the ideal situation, we would have measured the performance of the location service by starting a real-life search tree, that is, a search tree consisting of a large number of physical nodes distributed over a wide-area network, and measuring its performance under a real-life workload. Unfortunately, this approach was not possible for the following three reasons. First, the number of physical nodes required to run a real-life search tree was too large to handle by ourselves. Second, the characteristics of the network between the physical nodes were unknown. Third, the real-life workload of the location service (i.e., the number of invocations and their distribution over time and locations) was also unknown. Therefore, a different approach had to be taken.

8.4.1 Measurement Approach

In our alternative approach, we measured the performance of an individual physical node under several reasonably likely workloads. By focusing on the performance of a single physical node instead of the performance of the location service as a whole, we avoided both the problem of the large number of nodes and the problem of the unknown network characteristics. By examining the performance of the physical node under several reasonably likely workloads, we avoided the problem of the unknown real-life workload. We basically assumed that by examining reasonably likely workloads, we would cover any potential future workload. Note that this measurement of the performance of individual nodes can be seen as complementary to the simulations of Chapter 3 and Chapter 4 that examined the global interactions in the search tree.

We examined the different types of workloads by dividing them into separate **measurement cases**. Each measurement case examined a different type of behavior of an update or lookup operation at a physical node. We expected different types of behavior to result in different types of workloads for the physical node. Examples of measurement cases are: ”How long does it take to find a contact address in a physical node?” and ”How long does it take to insert a forwarding pointer at a physical root node?” By carefully choosing the structure of the search tree, we could examine each particular measurement case, as shown in the sections on the update and lookup measurement below.

Measuring the overall performance of a physical node by examining different measurement cases can be a problem in itself because there might be many different cases to consider for lookup and update operations. We therefore needed to introduce some simplifying assumptions to limit the number of cases to consider. The assumptions and the resulting measurement cases are described in the next two sections. As a first step to simplify the performance measurements, we considered the measurement cases for the update and lookup operations separately. This separation allowed us to ignore the problem of determining the real-life lookup-to-update ratio for object handles.

The shift of focus from the performance of the location service as a whole to the performance of individual physical nodes meant that the user view could no longer be measured directly since the duration of an individual operation (as experienced by a user) is in part determined by the network conditions. We could, however, still determine the time spent per operation in a physical node, and use that value to estimate the duration of individual operations in the location service as a whole. We discuss the consequences of the measurement results for the location service as a whole in Section 8.7.

Since it was difficult to measure the amount of time spent by each operation in a physical node, we did not measure this amount directly, but computed the amount of time instead, as the total duration of the measurement divided by the number of operations executed. This computation was possible since the implementation of the physical node was single threaded. We call the amount time spent in a physical node by an operation its *duration* in the node.

8.4.2 Physical Setup

To perform the measurements, we used the DAS-2 supercomputer at the Vrije Universiteit in Amsterdam. The DAS-2 is a supercomputer consisting of 72 independent PCs connected through a high-speed Myrinet network[1] and a normal fast Ethernet network. Every PC consisted of a dual 1-GHz Pentium-III system with 1 GByte of main memory and a local IDE hard disk with 20 GByte of storage. We used these disks to store the database, cache, and log of each physical node. The hard disk was formatted with the ext3 file system. For communication in the location service, we used the TCP/IP protocol over the fast Ethernet network. The PCs ran the Red Hat Linux 7.2 operating system, with a locally configured and compiled 2.4.19 kernel.

The measurement setup consisted of a small number of physical nodes that formed a search tree and a client program to generate update or lookup operations in that tree. To examine a measurement case, we ran each individual physical node, as well as the tree information service and the client program, on a separate PC in the supercomputer. The client program allowed multiple parallel operations to be started in the search tree. By varying the number of parallel operations, we could control the workload of the location service. By using several measurement runs, we could determine which workload resulted in a maximum throughput in a node without overloading the measurement setup.

To ensure real-life performance of the Berkeley database library, we filled the contact record database and location cache with a large number of extra object handles. These object handles were not used during the measurements, but did increase the size of the database files. The increased file size resulted in a slower, but more realistic, performance of the read and write operations on the databases.

[1] We did not use the high-speed Myrinet network in our measurements. The DAS-2 merely provided us with a couple of unused PCs connected through an independent and lightly-loaded network.

8.5 Measuring Update Operations

8.5.1 Determining Measurement Cases

To determine the measurement cases for update operations, we looked at their behavior at different types of nodes. We limited the number of measurement cases to consider with the following simplifying assumptions:

- The location service stores at most one contact address per object handle.
- The location service stores contact addresses only at leaf nodes.

These assumptions have the following consequences for the update operations:

- When a contact address is inserted, the leaf node always inserts the contact address, and the other nodes on the path from the leaf to the root node always insert a forwarding pointer.
- When a contact address is deleted, the leaf node always deletes the contact address, and the other nodes on the path from the leaf to the root node always delete their forwarding pointers.

The behavior of each of the nodes on the path from the leaf to the root node is therefore fixed by the assumptions:

- The leaf node receives an insert (or delete) request from a client, inserts (or deletes) the contact address, and requests the insertion (or deletion) of a forwarding pointer at its parent.
- One or more intermediate nodes receive an insert (or delete) request from a child node, insert (or delete) a forwarding pointer, and request the insertion (or deletion) of a forwarding pointer at their parent.
- The root node receives an insert (or delete) request from an (intermediate) child node, and inserts (or deletes) a forwarding pointer.

Since the behavior of the update operation is determined only by the type of node, we needed to consider only three types of nodes: leaf, intermediate, and root nodes. The measurement cases to consider for update operations consisted therefore of measuring the throughput and time distribution at these three types of nodes, as shown in Table 8.1.

We also assumed that the number of contact addresses stored in the search tree would be more-or-less stable during the measurement runs. This meant that during the measurements roughly the same number of insert and delete operations had to be executed. This steady-state assumption holds only for a long-running location service since the service is expected to grow in its start-up phase. Since any insert-to-delete ratio during the start-up phase would be a wild guess at best, we decided to focus on the steady-state situation only.

Table 8.1: Measurement cases for the update operation.

Case	Description
U1	Update operation executed at leaf node
U2	Update operation executed at intermediate node
U3	Update operation executed at root node

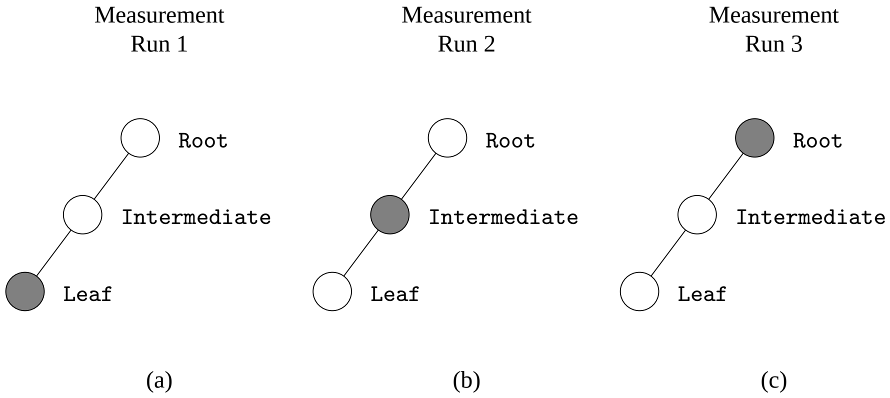


Figure 8.2: Search tree structures used in the three measurement runs for update operations. The (physical) node that was measured is shaded gray.

8.5.2 Measurement Methodology

To examine the three measurement cases, we used the minimal search tree that would allow us to measure the three types of nodes. The search tree consisted of a single leaf, intermediate, and root node, as shown in Figure 8.2. We used three independent measurement runs to measure the performance of the three types of nodes. In each of the three measurement runs, we measured the duration and time distribution of 5,000 update operations. Each update operation used an object handle that was randomly chosen out of a set of 20,000 object handles. To ensure that roughly the same number of insert and delete operations were executed 10,000 (i.e., 50%) of the 20,000 object handles were already stored in the location service at the start of a measurement run. The characteristics of the measurements are shown in Table 8.2.

To ensure that at most one contact address per object handle was stored in the location service, the client program maintained a table that kept track of which object handles had a contact address stored in the location service. The client program used this table to decide whether it should insert or delete a contact address for the randomly chosen object handle. If an address was present in the location service, the client program would delete the address; if no address was present, the client program would insert it.

Table 8.2: Characteristics of the measurement of update operations.

3	Measurement cases
3	Measurement runs
1	Search tree
1	Measurement scenario
5,000	Update operations
± 50%	Insert operations
± 50%	Delete operations
20,000	Object handles

Table 8.3: Measurement scenario for update operations. This scenario is used in all three measurement runs.

Step	Action	Description
1	Prepare	Insert 50% of all object handles.
2	Measure	Perform updates operations.

The measurement scenario for update operations consisted of the same two steps for all three measurement runs. These steps are shown in Table 8.3. In the first step, we prepared the search tree by inserting 50% of all the object handles used in the experiment. In the second step, we measured the performance of update operations by performing 5,000 update operations on randomly chosen object handles. We measured how long the update operations took in total and measured how much time was spent in the various layers and modules in total. The throughput is the number of operations divided by the total duration of the measurement run. The duration of an individual operation is the total duration of the measurement run divided by the number of operations.

To ensure that we measured the maximum throughput of a node, we needed to ensure that the measured node was the bottleneck in the system. Otherwise, we might actually have measured the maximum throughput of one of the other two nodes, given the pipeline nature of executing operations in the location service. We avoided this problem by artificially lowering the amount of work per operation in the other two nodes in the search tree, for example, by using asynchronous writes to disk or by using a message log in main memory at these nodes.

We expected that in normal operations a physical node would usually read its contact records directly from disk, instead of from the file system cache, since the number of contact records stored by a node is normally quite large. To ensure that the contact record database would behave in the same way in our measurement setup, we therefore needed to minimize the caching effects of the file system. We minimized the caching effects by storing significantly more object handles in the database than the number of operations,

Table 8.4: Throughput and duration of an individual update operation in different types of physical nodes.

Case	Description	Throughput	Duration
U1	Leaf node	9.2 updates/s	108.1 ms
U2	Intermediate node	16.7 updates/s	59.6 ms
U3	Root node	17.3 updates/s	57.6 ms

Table 8.5: Time spent per update operation in the different layers and modules (as depicted in Figure 8.1) for the three types of physical nodes.

Layer / Module	Leaf Node	Intermediate Node	Root Node
Low-level	0.7 ms	0.8 ms	0.2 ms
Messenger	0.9 ms	0.7 ms	0.4 ms
RPC (incl. log)	48.2 ms	0.5 ms	0.3 ms
Algorithm	0.6 ms	0.5 ms	0.4 ms
CRDB	57.7 ms	57.1 ms	56.3 ms
Cache	0.0 ms	0.0 ms	0.0 ms
Total	108.1 ms	59.6 ms	57.6 ms

thereby minimizing the chance of reusing one of the randomly chosen object handles in an operation. For our measurements, we chose to use four times as many object handles as update operations.

8.5.3 Results

Table 8.4 shows the combined results of the throughput and of the duration measurements of update operations in the three measurement cases. The table shows that the leaf node is only half as fast as the other two nodes. The root and intermediate node have almost the same performance. These results are to be expected, and can easily be explained by Table 8.5.

Table 8.5 shows the time distribution per update operation over the various layers and modules in a physical node, as depicted in Figure 8.1. The table shows that most of the time is spent performing disk operations, either by reading and writing contact records in the database (in the CRDB module), or by saving request messages in the persistent message log (in the RPC layer). Since the leaf node has to do both, it is only half as fast as the other nodes. The root node is slightly faster than the intermediate node since it has less communication to do.

8.6 Measuring Lookup Operations

Since the behavior of lookup operations is much less influenced by the type of node its runs on than the behavior of update operations, we took a different approach to find the measurement cases for the lookup operation. Instead of looking at the behavior of the lookup operation at different types of nodes, we examined the different steps taken by the lookup operation to find a contact address.

8.6.1 Determining Measurement Cases

We limited the number of measurement cases to consider for the lookup operation with the following simplifying assumptions:

- Only a single contact address is stored in the search tree per object handle.
- Only a single contact address is looked up per lookup operation.

These assumptions have the following consequences for the lookup operation:

- The lookup operation can always find a contact address.
- The lookup operation is always ready after the first contact address is found.

In Section 4.4 we described that the lookup operation uses five steps to find contact addresses. Since every lookup operation always finds a single contact address during the measurements, one of these five steps is responsible for retrieving the contact address. These five steps thereby become five measurement cases:

i. The contact address is found locally at the physical node.

ii. The contact address is found by following a local cache pointer.

iii. The contact address is found by following a forwarding pointer.

iv. The contact address is found by following a remote cache pointer.

v. The contact address is found by going to the parent node.

We also needed to consider the fact that the location cache might have invalid cache entries, that is, references to nodes that no longer store a contact address. This was potentially a problem since the cache could have more than one invalid cache entry per object handle. To avoid having to deal with potentially large numbers of measurement cases involving increasing numbers of incorrect entries in the location cache, we also introduced a third simplifying assumption for the lookup operation:

- A physical node has at most one invalid cache entry per object handle.

Table 8.6: Measurement cases for the lookup operation.

Case	Description
L1	Local contact address
L2	Via local cache pointer
L3	Via forwarding pointer
L4	Via forwarding pointer, after an invalid cache entry
L5	Via remote cache pointer
L6	Via parent
L7	Via parent, after an invalid cache entry

Since the lookup operation continues searching for a contact address after using an invalid cache entry, the invalid cache entry must always be examined before or during the step that actually retrieves the contact address. The extra cases created by an invalid cache entry can thus be considered variants of case ii–v.

When considering these four invalid-cache-entry variants, the variants of case ii and case iii are seen to be very similar, as well as those of case iv and case v. We consider case ii and case iii to be similar because for both cases the lookup operation accesses the contact record database and the local location cache, before performing the RPC that retrieves the contact address. For case iv and case v, the lookup operation accesses the contact record database, the local location cache, and the remote location cache, before performing the RPC that retrieves the contact address. Because of these similarities, we decided to add only two new measurement cases for invalid cache entries:

vi. The contact address is found by following a forwarding pointer, after trying an invalid local cache entry (variant of case iii).

vii. The contact address is found by going to the parent node, after trying an invalid remote cache entry (variant of case v).

In total, we thus had seven measurement cases to consider for the lookup operation. The seven cases are summarized (and slightly reordered) in Table 8.6.

8.6.2 Measurement Methodology

To examine the seven measurement cases for the lookup operation, we used three measurement runs and two search trees, as shown in Figure 8.3. Per search tree, we used a single measurement scenario. To measure the performance of a physical node in a particular measurement case, the client program invoked 50,000 lookup operations which each operation looking up a single object handle. The characteristics of the lookup operation measurements are shown in Table 8.7.

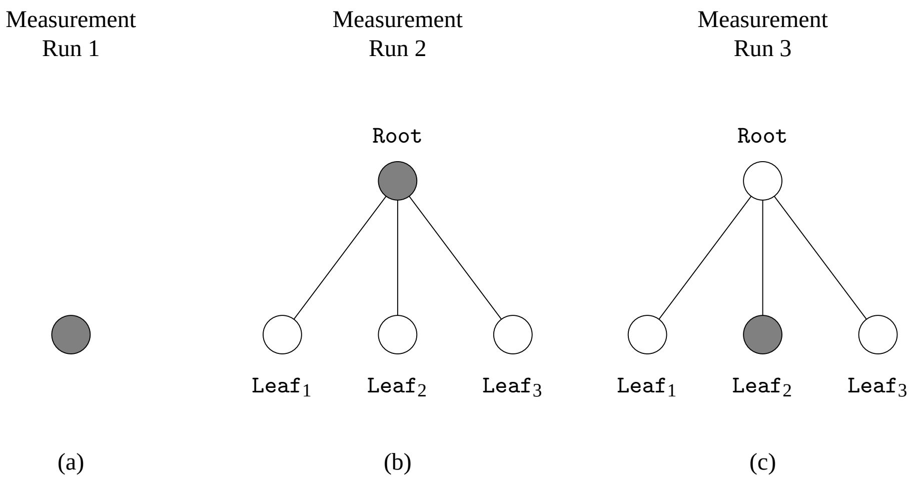


Figure 8.3: Search tree structures used in the three measurement runs for lookup operations. The (physical) node that was measured is shaded gray.

Table 8.7: Characteristics of the measurement of lookup operations.

7	measurement cases
3	Measurement runs
2	Search trees
2	Measurement scenario
50,000	Lookup operations
50,000	Object handles

Table 8.8: Measurement scenario 1 for lookup operations. This scenario is used in measurement run 1.

Step	Action	Description
1	Prepare	Insert all object handles at the leaf node.
2	Measure	Lookup all object handles at the leaf node.

As with the measurement of update operations, we had to ensure that only the measured node was the bottleneck in the system. This was obviously not a problem for measurement run 1 since it used a search tree consisting of only a single node. To speed up the other nodes in the search tree in run 2 and run 3, we disabled their location caching code and emptied their contact record database.

As explained in Chapter 4, the lookup procedure inserts a new location in its location cache whenever it finds a contact address at another node. Since this cache update is of no concern to the user that invoked the lookup operation, this update can be performed after the RPC reply is sent back to the caller. The cache update is thus not part of the duration of the lookup operation as experienced by the user. It is, however, measured in the total duration of the measurement run, and thus the duration of individual lookup operations. We therefore had to correct the duration of lookup operations in a node by subtracting the total cache update time from the total duration. We added an extra timer to the physical node prototype to measure the total time used to update the location cache.

Measurement Run 1

The goal of the first measurement run was to examine measurement case L1, that is, find a contact address at the local physical node. For this run, we used a search tree consisting of a single physical node, as shown in Figure 8.3(a). The measurement scenario for this run consisted of two steps, as shown in Table 8.8. Step 1 prepared the search tree by inserting all object handles in the single node. Step 2 performed the actual measurement by letting the client program invoke lookup operations for all object handles and measuring the performance.

Measurement Run 2

The goal of the second measurement run was to examine measurement cases L2–L4. In these cases, the contact address is found, either by following a local cache pointer or by following a forwarding pointer. To find a node that exhibited the behavior associated with these cases, we used a search tree consisting of a single (physical) root node and three leaf nodes (each consisting of a single physical node), as shown in Figure 8.3(b). Since the root node stores a forwarding pointer when a contact address is stored in one of the leaf nodes and a local cache pointer when a contact address is found in one of the leaf nodes,

Table 8.9: Measurement scenario 2 for lookup operations. This scenario is used in measurement runs 2 and 3.

Step	Action	Description
1	Prepare	Insert all object handles at node $\texttt{Leaf}_1$.
2	Measure	Lookup all object handles at node $\texttt{Leaf}_2$.
3	Measure	Lookup all object handles at node $\texttt{Leaf}_2$.
4	Prepare	Delete all object handles at node $\texttt{Leaf}_1$.
5	Prepare	Insert all object handles at node $\texttt{Leaf}_3$.
6	Measure	Lookup all object handles at node $\texttt{Leaf}_2$.

the behavior associated with cases L2–L4 is exhibited by the root node. We measured in this run therefore the performance of the root node.

The measurement scenario for run 2 is shown in Table 8.9. Steps 1, 4, and 5 were used to prepare the search tree, and steps 2, 3, and 6 performed the actual measurements. The measurement steps consisted of performing a lookup operation for every object handle. Using every object handle only once minimized the caching effects of the file system. In these steps, the lookup operations were always started at node $\texttt{Leaf}_2$.

Step 1 prepared the search tree for measurement steps 2 and 3 by inserting at node $\texttt{Leaf}_1$ a contact address for every object handle. Step 2 measured the performance of finding a contact address via a forwarding pointer (i.e., case L3) by starting lookup operations at node $\texttt{Leaf}_2$. As a side effect of the lookup operation, cache entries were inserted in this step in the local location cache of the root node. No cache entries were inserted in node $\texttt{Leaf}_2$ since its location cache was disabled, as described above. Step 3 measured the performance of finding a contact address via a local cache pointer (i.e., case L2) by using the local cache entries inserted in step 2.

The remaining case was finding a contact address via a forwarding pointer, after using an invalid cache entry (i.e., case L4). This case used invalid entries in the local location cache. To create such invalid entries, we moved the contact addresses from node $\texttt{Leaf}_1$ to node $\texttt{Leaf}_3$. This movement was done in step 4 and step 5. These steps deleted the contact addresses at $\texttt{Leaf}_1$ and inserted them at $\texttt{Leaf}_3$. Since the entries in the location cache at root node still pointed to node $\texttt{Leaf}_1$, step 6 measured the performance of physical nodes with invalid cache entry caches at the root node, as needed for this case.

Measurement Run 3

The goal of the third measurement run was to examine measurement cases L5–L7. In these cases, the contact address is found either by following a remote cache pointer or by going to the parent node. For this run, we used the same search tree and the same measurement scenario as measurement run 2. The main difference with the measurement run 2 is the node that is measured. In this run, we examined the performance of node $\texttt{Leaf}_2$ since that node exhibits the behavior associated with cases L5–L7.

Step 1 prepared the search tree for measurement steps 2 and 3 by inserting at node $\texttt{Leaf}_1$ a contact address for every object handle. Step 2 measured the performance of finding a contact address by going to the parent node (i.e., case L6) by starting lookup operations at node $\texttt{Leaf}_2$. Again, as a side effect of the lookup operation, cache entries were inserted in this step in the remote location cache of node $\texttt{Leaf}_2$. No cache entries were inserted in the root node since its location cache was disabled, as described above. Step 3 measured the performance of finding a contact address via a remote cache pointer (i.e., case L5) by using the remote cache entries in node $\texttt{Leaf}_2$.

The remaining case was finding a contact address by going to the parent node, after using an invalid cache entry (i.e., case L7). This case used invalid entries in the remote location cache. To create such invalid entries, we moved the contact addresses from node $\texttt{Leaf}_1$ to node $\texttt{Leaf}_3$ in this run as well. This movement was done in step 4 and step 5. Step 6 measured the performance of physical nodes with invalid cache entry caches at node $\texttt{Leaf}_2$, as needed for this case, since the entries in the location cache at node $\texttt{Leaf}_2$ still pointed to node $\texttt{Leaf}_1$.

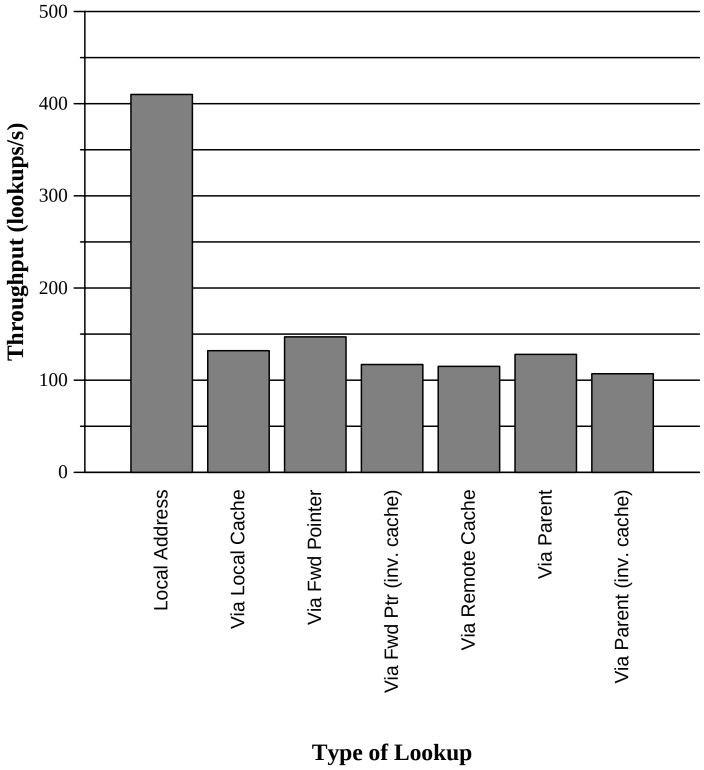
8.6.3 Results

Figure 8.4 shows the maximum throughput of lookup operations in a physical node. The measurement cases can be divided into two groups based on the performance. The first group consists of only the case where the lookup operation finds a contact address directly in the local node (i.e., L1). In this case our prototype can handle about 410 lookup operations per second.

The second group consists of the other cases, which were significantly slower. The slowest case is case L7, where a lookup operation has to handle an invalid cache entry and has to use the parent node to find a contact address. In this case our prototype can handle about 107 lookup operations per second. Figure 8.5 shows the duration of lookup operations for the different cases. These numbers include the correction for inserting cache entries, as mentioned above, and show the same kind of result as the numbers for the throughput, although the difference between the cases is less pronounced.

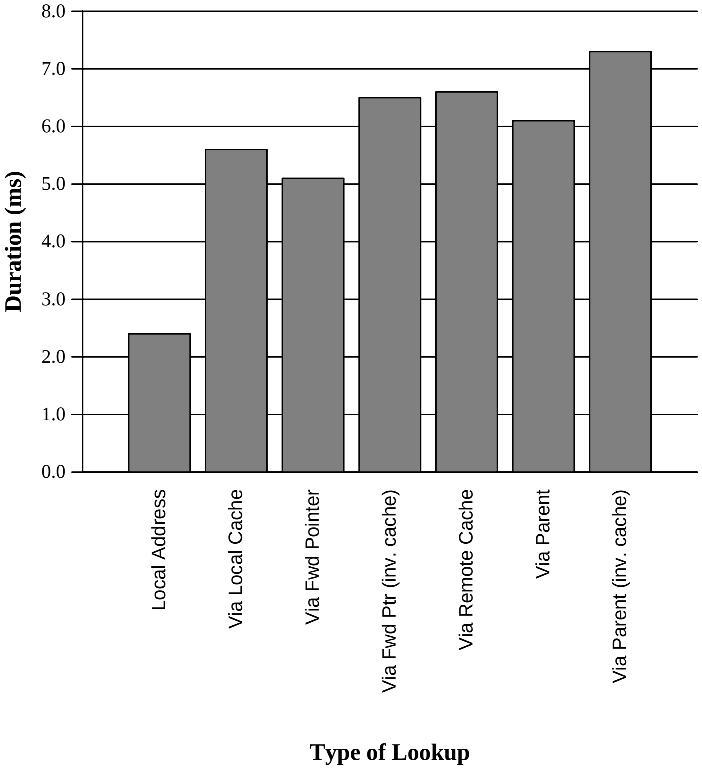
The bar graph in Figure 8.6 allows us to easily compare the engineering view for the different measurement cases. In the figure, the cache module has an extra white bar which indicates the amount of time spent on the cache update. This time is spent in the node, but it is not part of the duration as experienced by a user.

The figure clearly shows that cache access times are the main reason why case L1 is so much faster than the other measurement cases. It also shows that for cases L2–L7, the time distribution is almost the same. The main anomaly in the figure is that the algorithm layer is 1 ms slower in cases L5–L7, while the node has to execute only a few extra lines of Java code for these cases. The lines do not explain the significant increase in the amount of time used since they only determine that steps 1–3 of the lookup procedure do not apply. Unfortunately, we do not have an explanation for this anomaly.



Case	Description	Throughput
L1	Local contact address	410 lookups/s
L2	Via local cache pointer	132 lookups/s
L3	Via forwarding pointer	147 lookups/s
L4	Via fwd. ptr. (inv. cache)	117 lookups/s
L5	Via remote cache pointer	115 lookups/s
L6	Via parent	128 lookups/s
L7	Via parent (inv. cache)	107 lookups/s

Figure 8.4: Throughput (i.e., the number of lookup operations per second) of different cases for the lookup operation measurement.



Case	Description	Duration
L1	Local contact address	2.4 ms
L2	Via local cache pointer	5.6 ms
L3	Via forwarding pointer	5.1 ms
L4	Via fwd. ptr. (inv. cache)	6.5 ms
L5	Via remote cache pointer	6.6 ms
L6	Via parent	6.1 ms
L7	Via parent (inv. cache)	7.3 ms

Figure 8.5: Duration of different types of lookup operations in a physical node (critical path only).

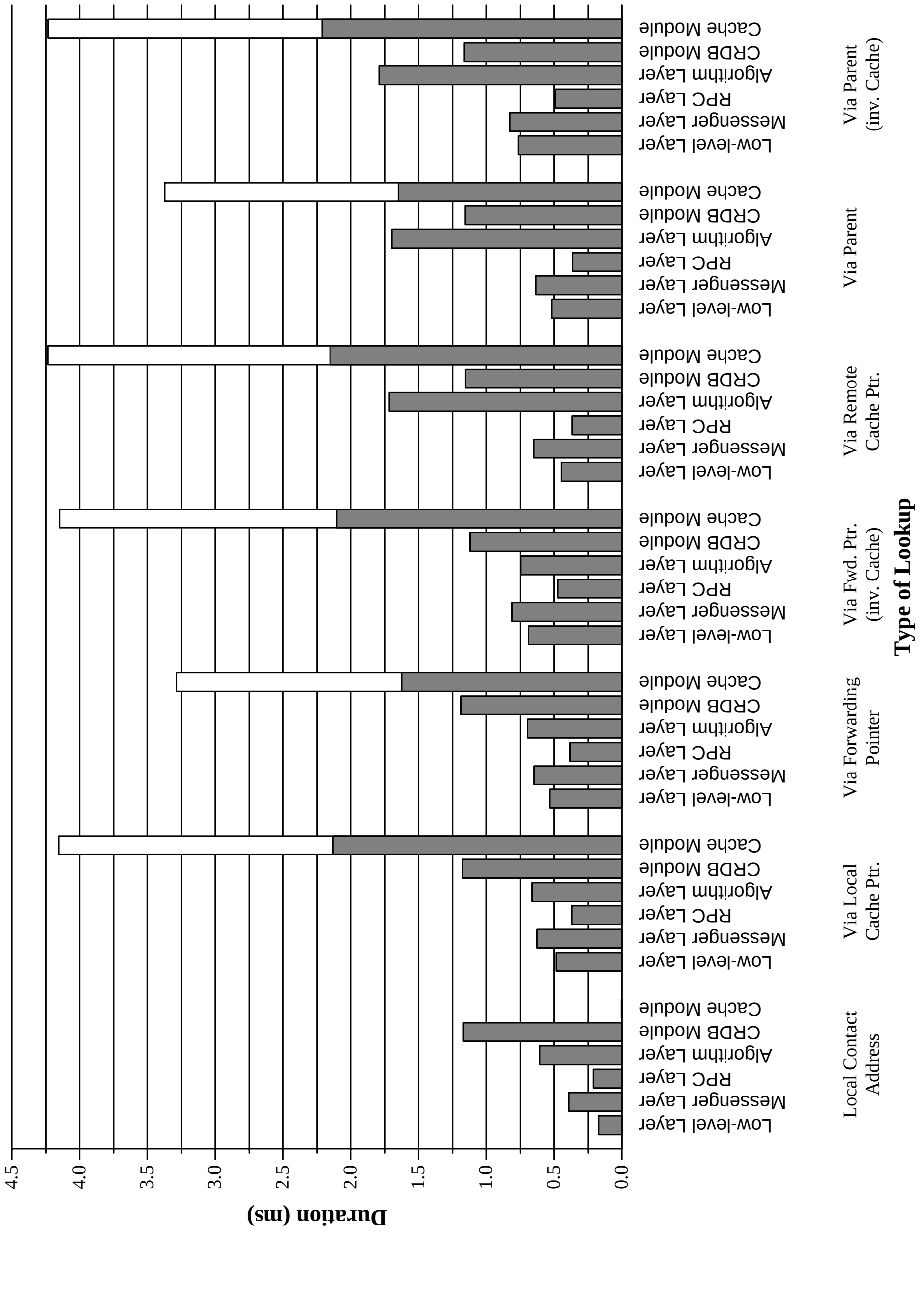


Figure 8.6: Duration of different parts of the lookup operation in a physical node for different types of lookup operations. The critical path is shaded gray.

8.7 Discussion of Results

The main question examined in this chapter is whether an efficient physical node can be implemented. To answer this question, we have to determine whether the performance of the physical nodes allows the location service to be scalable. We make this determination by examining the performance numbers of the last two sections in the context of a complete location service.

User-view Assessment

In our assessment of the performance numbers, we first consider the user view, that is, the duration of update and lookup operations in the location service as a whole. As we indicated before, it is, unfortunately, difficult to state what typical update and lookup operations are in the location service. We therefore examine the performance numbers by using them to estimate only the worst-case performance of update and lookup operations in a typical example search tree. Our goal in this user-view assessment is to show is that the worst-case performance of update and lookup operations is within acceptable bounds.

An update operation consists at most of executing update procedures at all the nodes on the path from the leaf node to the root, that is, at the leaf node, some intermediate nodes, and the root node. The worst-case scenario for an update operation is therefore performing update procedures at all these nodes. We examine the performance in a four-level search tree, as used in Chapters 3 and 4. The performance of the leaf node was captured by measurement case U1, the performance of the two intermediate nodes was captured by measurement case U2, and the performance of the root node was captured by measurement case U3. In the example search tree, the time spent in the nodes of the search tree for the worst-case scenario is:

$$108.1\,\text{ms}\,(\text{case U1}) + 2 \times 59.6\,\text{ms}\,(\text{case U2}) + 57.6\,\text{ms}\,(\text{case U3}) = 284.9\text{ ms}$$

For the total duration of the update operation in the worst-case scenario, we need to add the communication time between the four nodes. If our location-aware load distribution scheme is effective, the physical nodes are near each other, and the total communication time between nodes can be as low as a couple of milliseconds. If the load distribution scheme is not effective, however, the physical nodes can be located far away, and the total communication time between nodes can easily rise to several hundred milliseconds. Given that there are three remote RPCs involved in the example search tree, and assuming a round-trip between 5 and 100 ms time per RPC, we expect the total duration of the update operation in the worst-case scenario to be roughly between 300 and 600 ms.

To evaluate the estimated total duration of update operations (i.e., 300–600 ms), we examine the context where these operations are used. We are mainly interested in duration of insert operations since the completion of an insert operation signifies the moment the contact address is available to all users and thereby the moment the object handle can safely be distributed. In contrast, the completion of the delete operation only signifies the disappearance of a contact address, which is less interesting. A new contact address is inserted in the location service when a new replica is created. The creation of a replica at a new location can be considered a heavy operation since it usually involves the (wide-area) transfer of state and program code. Since these transfers can last in the order of seconds themselves, the duration of the insert operation in the worst-case scenario is acceptable.

For lookup operations, a general, worst-case scenario does not exist since we can always increase the amount of work by having more contact addresses in the search tree, by searching for more contact addresses, or by having more invalid entries in the location caches of the tree nodes. If we stick with the simplifying assumptions for lookup operations from the previous section, however, the worst-case scenario for lookup operations is easily given. It consists of the lookup operation starting at a leaf node, visiting each node on the path from the leaf to the root node, and following forwarding pointers down to the leaf node that stores the contact address, while examining a single invalid cache entry at each node visited.

In this limited worst-case scenario, the lookup operation consists of executing the lookup procedure at two nodes at the leaf and intermediate levels of the search tree and at one node at the root level. While going up the search tree via the parent node, the lookup operation behaves according to measurement case L7, and while going down the search tree by following forwarding pointers, the lookup operation behaves according to measurement case L4. At the leaf node that stores the contact address, the lookup operation behaves according to measurement case L1. If we consider a four-level search tree, this means that the time spent in nodes is:

$$3 \times 7.3\,\text{ms}\,(\text{case L7}) + 3 \times 6.5\,\text{ms}\,(\text{case L4}) + 2.4\,\text{ms}\,(\text{case L1}) = 43.8\text{ ms}$$

For the total duration of the lookup operation in the limited worst-case scenario, we need to add the communication time between the seven nodes. Once again, the total communication time can be as low as a couple of milliseconds and grow to several hundred milliseconds. For the lookup operation, however, there is communication between seven nodes involved. Assuming again a round-trip between 5 and 100 ms time per RPC, we therefore expect the total duration of the lookup operation in the limited worst-case scenario to be roughly between 75 and 650 ms.

Luckily, if our location caching scheme is effective, the lookup operation visits only two nodes, as discussed in Chapter 4. The first node is the leaf node that has a (valid) remote cache pointer (i.e., measurement case L5); the second node is the node that actually stores the contact address (i.e., measurement case L1). The time spent in nodes for a lookup operation is in this case significantly less:

$$6.6\text{ ms (case L5)} + 2.4\text{ ms (case L1)} = 9.0\text{ ms.}$$

In this case, the communication between only two nodes needs to be added. Assuming a round-trip between 5 and 100 ms time for a single RPC, we therefore expect the total duration of the lookup operation in this case to be roughly between 15 and 110 ms.

To evaluate the performance of the lookup operation, we compare it to the performance of name resolution of DNS. Depending upon the effectiveness of our location caching scheme, a lookup operation in the Globe location service lasts roughly between 15 and 650 ms. Since the duration for successful name resolution in DNS falls roughly between several milliseconds and several seconds, presented for instance in [Jung et al., 2001], we consider the lookup performance of our location service acceptable.

System-view Assessment

Next, we consider the system view, that is, the number of update or lookup operations a physical node can support. The measured throughput for update operations is, unfortunately, lower (i.e., worse) than the numbers we assumed in our calculations in Chapter 3. For example, if the logical root node were to receive on average an update operation for all objects once a month, the logical root node would receive 3.9×10^5 update operations per second. To support this number of operations with the prototype implementation, we need $3.9 \times 10^5/17.3 \approx 22,300$ physical nodes to implement the logical root node. This number is quite large, but still within the expected boundaries, as given in Chapter 3.

The measured throughput for lookup operations is higher (i.e., better) than the throughput we assumed in Chapter 3. For example, if the logical root node were to receive a lookup operation for all objects once a month, the logical root node would receive 3.9×10^5 lookup operations per second. To support this number of operations with the prototype implementation, we need $3.9 \times 10^5/117 \approx 3,300$ physical nodes to implement the logical root node. This number is also large, but within the expected boundaries, as given in Chapter 3.

Engineering-view Assessment

The engineering view clearly shows that for both update and lookup operations, the persistent storage (i.e., reading and writing to the hard disk) is the main performance bottleneck. This problem is the most pronounced for update operations since they require synchronous write operations for database updates and message logging.

An obvious hardware solution to this problem is the use of nonvolatile RAM (NV-RAM). For instance, the persistent message log can easily be stored completely in NV-RAM since the log is unlikely to grow large. With this approach we avoid one synchronous disk write (to add the message to the log) and one asynchronous disk write (to remove the message from the log) per update operation. NV-RAM can also be used to collect several database updates persistently in main memory, and write them back to disk asynchronously. A similar method is described for NFS in [Baker et al., 1992].

A software solution to the performance problem is collecting disk write operations, and perform them synchronously as a group. This aggregation improves disk write performance by better use of the bandwidth to the disk, but delays the completion of individual update operations. To what extent such aggregation can be effective remains an open question.

Chapter 9

Related Work

This chapter compares the Globe location service to related work. The criteria on which the related work is chosen are similarity in the functionality provided, such as support for mobility and replication, and similarity in the techniques used. Based on these grounds, we chose the following six categories of related work: naming and directory services, home-location based systems, search-tree based systems, pointer-chain based systems, uniform resource names (URNs), and peer-to-peer (P2P) systems. Within every category, we discuss one or two example systems, and compare these to the Globe location service. We compare the related work using the issues raised in Chapters 2–7: replication support, mobility support, basic architecture, scalability, fault tolerance, security, and management (adaptability). A more general overview of research in the area of location services can be found in [Pitoura and Samaras, 2001].

9.1 Naming and Directory Services

The primary goal of the location service is to provide location transparency. Historically, research in the area of location transparency falls in the naming services category. Examples of such services are DNS [Mockapetris, 1987; Albitz and Liu, 1992], the Tilde naming system [Comer et al., 1990], work by Cheriton and Mann [Cheriton and Mann, 1989], and work by Lampson [Lampson, 1986]. In this section we consider wide-area naming services and wide-area directory services to be a *single* category. Even though wide-area directory services, such as those based on X.500 [Radicati, 1994] or LDAP [Loshin, 2000], provide more powerful search facilities than wide-area naming services, they are still quite similar in the technology used. In this section we use the Domain Name System (DNS) as our example system.

Summary

DNS is part of the Internet structure, and is used to give human-friendly names to Internet nodes and e-mail destinations. As a naming service, DNS has a strong focus on allowing humans to select the right node or destination (i.e., human friendliness). DNS is, however, also used as a location service in several content distribution networks (CDNs), such as Akamai [Akamai Technologies, Inc., 2002].

We can distinguish a logical structure and a physical structure in DNS. The logical structure of DNS concerns its tree-shaped name space. This name space is similar to the name spaces found in present-day file systems, and differs mostly in the terminology used and in the syntax of names. For instance, in DNS directories are called **domains** and names are called **domain names**. Syntactical differences include the order in which names are interpreted (i.e., right-to-left instead of the left-to-right usually found in file systems) and the use of the dot character (".") instead of the (backward) slash character ("/" or "\") to divide a domain name into a sequence of labels.

The physical structure of DNS consists of a large set of name servers. Each name server is assigned a part of the name space to support. The assignment is done as follows. First, the overall name space is partitioned into **zones** by grouping sets of related domains; then, each name server is assigned one or more zones to support. A name server supports a zone by persistently storing its contents and allowing other hosts (i.e., clients and servers) to query the information contained in the zone.

Traditionally, DNS servers support two protocols: a name resolution protocol and a zone replication protocol. Changes to the domain information in a zone were made manually, by simply editing the configuration files containing the zone information. In recent years, however, an update protocol has been added to DNS, which enables clients to make simple modifications to the zone information remotely. The authors of [Huck et al., 2002] go further and propose to extent DNS with a management system that is designed to significantly reduce the amount of work required for the (re)configuration of DNS servers. In this system each name server is assigned a management agent that is responsible for maintaining the configuration of the name server.

Name resolution in DNS is closely tied to the structure of a domain name. A domain name consists of a sequence of labels. The sequence of labels determines a path of domains in the tree-shaped name space, starting at the root domain. Conceptually, name resolution consists of visiting, in turn, the domains associated with the labels in the sequence. In practice, this means visiting the name servers responsible for the zones encompassing the domains. To improve the efficiency of name resolution, DNS makes extensive use of caching. Name servers cache both intermediate and end results of the name resolution process.

To improve fault tolerance and availability, DNS supports the replication of zone information among multiple name servers. The name servers use a master-slave protocol to propagate changes in a zone. Changes are made at the master name server either manually or through the update protocol, and slave name servers periodically retrieve the updated state of the zone using a special zone transfer protocol.

Several CDNs, such as Akamai, have introduced an extension to DNS that improves the locality of the result of the name resolution process. Normally, when faced with a replicated host, that is, a domain name with more than one IP address, the name resolution process picks an IP address of the set of available addresses in a round-robin fashion. The extension, however, selects an IP address of the replicated host that is near the client that requested the name resolution. To determine which IP address of the domain name is near, the extension uses the IP address of the client. The exact method Akamai uses to choose an IP address is left unspecified, but it does take the topology and current use of the Internet into account [Leighton and Lewin, 2000].

Evaluation

An important part of DNS, as with any name service, is its support for human-friendly naming. This aspect of DNS, however, falls outside the scope of our evaluation since human-friendly naming is not the responsibility of the Globe location service, as described in Chapter 1.

DNS has support for replicated hosts, that is, domain names with more than one IP address. However, this support normally does not include exploiting locality, and an extension, as described above, is thus needed. DNS has no support for frequent mobility. It can behave like a home-location based system by allowing the IP address associated with a mobile host to be changed. Frequent address changes of a mobile host do, however, preclude the use of replication and caching techniques.

Even though DNS and the Globe location service both use a distributed search tree to deal with scalability issues, the purpose of the search tree is actually significantly different in both systems. While in DNS, the search tree is used to organize large numbers of domain names and make them available in a human-friendly fashion, the search tree in the Globe location service is used to represent distances in the underlying network in order to exploit locality.

DNS has proven it can store a large number of names[1], and thus clearly has numerical scalability. DNS also provides geographical scalability through its use of caching to prevent wide-area communication. The extension mentioned before can provide even more locality by carefully selecting the answers (i.e., IP addresses) of the name-resolution process. Recently, security features have been introduced in DNS to protect against unauthorized changes of the name-to-address mapping. Most hosts on the Internet, however, still use the insecure versions of the DNS protocol.

9.2 Home-location based Systems

In a home-location based system, each mobile object has an identifier that identifies a server in the network that is responsible for storing the current location of the mobile

[1] 1.5×10^8 names, as of January 2002 [Internet Software Consortium, 2002].

object. This server is called the object's **home location**. When a client wants to communicate with the mobile object, it queries the home location to retrieve the current location of the object, and when the mobile object moves, it sends an update request to its home location to update its current location. Note that this approach is not centralized. Different mobile objects can have different servers as a home location.

The two examples of home-location based systems described in this section are the HLR/VLR system [Mohan and Jain, 1994], found in current mobile-phone systems, and Mobile IP [Perkins, 2002], which is the current proposal for supporting host mobility in the Internet. Both example systems have extended the basic home-location based architecture, and are therefore not home-location based location services in a strict definition of the term.

9.2.1 HLR/VLR Systems

Summary

The HLR/VLR architecture is currently used in the location services of several (cellular) mobile-phone systems. The two most notable examples of its use are IS-41 (Interim Standard 41) systems, which are used in North America, and GSM (Global System for Mobile Communication) systems, which are used primarily in Europe.

The cellular mobile-phone architecture can be divided into a mobile and a fixed part. The mobile part is the most visible and consists of mobile phones capable of wireless communication. The fixed part is less visible and consists of the support structure distributed across the area where mobile phones can be used. The support structure divides this area into small, disjoint cells. Every cell has a (fixed) base station capable of wireless communication. The mobile phone communicates using wireless communication with the base station of the cell in which it currently resides and from there on using normal wireline based communication.

The main mobility problem in cellular mobile-phone systems is determining in which cell a mobile phone currently resides. This information is needed to correctly route data packets for the mobile phone to the base station of the cell where the mobile phone currently resides. It can also be used for other services, such as locating people in emergency situations.

The fixed part of the mobile-phone architecture can be divided further into three parts: the **fixed** network, the **access** network, and the **intelligent** network. The fixed network is basically (part of) the normal public telephone switching network (PTSN). The access network consists of the transmitters and related hardware of base stations that provide the wireless communication. The intelligent network, finally, is used for all signaling communication, such as setting up and tearing down connections in the (mobile) phone system. For example, when a person on a fixed line calls a person on a mobile phone, the intelligent network connects the two using the fixed network for the first part of the connection and the access network for the second part.

The HLR/VLR architecture is part of the intelligent network, and uses two types of hosts — called “registers” — to record where a mobile phone is located. Each mobile phone has an associated **home-location register** (HLR) that stores its current location. Specifically, it records the **registration area** in which a mobile is located. A registration area is simply a combination of several adjacent cells. This basic home-location based approach is extended with the **visitor-location register** (VLR). The VLR records per registration area which mobile phones are present in that area.

When a mobile phone (i.e., the caller) calls another mobile phone (i.e., the callee), the current location of the callee is looked up in two steps. First, the mobile-phone system looks in the VLR of the caller to see if the callee is present in the same registration area as the caller. If so, a connection can be setup quickly. In this way, the VLR provides a form of locality for lookup operations. Otherwise, the mobile-phone system looks in the HLR of the callee to find its current location. If the callee is present in the area covered by the mobile-phone service, its location will be known by its HLR, and a connection can be set up.

When a mobile phone moves from a cell of one registration area to a cell of a different registration area, three actions need to be taken. First, the presence of the mobile phone needs to be inserted in the VLR of the new registration area. Second, the HLR of the mobile phone needs to be updated to reflect the current location of the mobile phone. Third, the presence of the mobile phone needs to be removed from the VLR of the old registration area. Movements between cells of the same registration area are handled by the registration area itself.

The efficiency of the HLR/VLR system can be improved by several extensions to the basic architecture. An obvious extension is the caching of mobile phone locations at the VLR. Using this form of caching the mobile-phone system can avoid communication with the HLRs of the (cached) mobile phones, and thereby shorten connection setup times. The choice to cache the location of a specific mobile phone is based upon its local call-to-mobility ratio (LCMR). A high value for the LCMR means that the mobile phone is frequently called from the current registration area, and caching will thus be effective.

A different extension is the use of forwarding pointers at the VLR to shorten the duration of update operations. With this extension, the mobile-phone system leaves a forwarding pointer at the old registration area of the mobile phone, instead of updating the location of the mobile phone at the home location. The mobile-phone system can then update the HLR in a lazy fashion. This improvement comes at the cost of an increased duration of a lookup operation, and should thus be used only if the call-to-mobility ratio (CMR) is low (i.e., the mobile phone is not frequently called).

Evaluation

Location services based on the HLR/VLR architecture are designed to support mobility only. There is no concept of replication (i.e., multiple mobile phones with the same number). There is some similarity between the search tree of the Globe location service and the HLR/VLR architecture. In fact, the HLR/VLR architecture can be considered a two-layer search tree with the set of HLRs forming a logically partitioned root node and the set of VLRs forming its leaf nodes.

Since the HLR/VLR structure is currently in use in mobile-phone systems, the architecture is obviously effective on a wide-area scale. The HLR/VLR architecture can easily scale further in the number of supported mobile phones since that simply means the use of more HLRs. It is, however, not obvious how well the HLR/VLR architecture can scale further geographically. The basic architecture provides some lookup locality through its use of the VLR, but it is not clear whether it provides enough locality to support a worldwide system. Furthermore, the basic architecture provides no locality for update communication.

9.2.2 Mobile IP

Summary

Mobile IP is a home-based routing system designed to support the connectivity of mobile Internet hosts. Since the current IPv4-based Internet is already fully deployed, the design goals of Mobile IP center around backward compatibility with the existing transport layer protocols, applications, and infrastructure. To support this compatibility, the Mobile IP protocols are an extension to the IPv4 standards that leave the existing functionality in place. Since the new IPv6-based protocols are not yet fully deployed, most techniques developed for IPv4 have also been integrated into the basic IPv6 standards. The Mobile IP protocols are developed by the Mobile IP working group of the IETF [Mobile IP, 2002].

Mobile IP solves the problem of the dual use of the IP address of a host for both routing and host identification. Since the contents and structure of IP addresses are used for routing, IP addresses are assigned in such a way as to enable efficient routing through the underlying network. Unfortunately, this means that when an Internet host changes its location in the network, it also needs to change its IP address to remain accessible by other nodes. On the other hand, (pairs of) IP addresses are also used to identify communication channels between hosts at the transport layer. Since present-day applications require these channels to remain in existence, that is, even when a mobile host changes its location, we explicitly do not want IP addresses to change. IP addresses should thus on the one hand be *location dependent* to enable routing and on the other hand be *location independent* to support durable communication channels.

Mobile IP solves the dual-use problem at the network layer by assigning two IP addresses to the **mobile host**, the basic mobile entity in Mobile IP. The first address is the normal, fixed IP address that is used for host identification and routing when the host is at its home location, The second is a temporary **care-of address** that is used to route network packets to the mobile host when it is roaming the network. Other nodes (called **correspondent hosts**) in the network are unaware of the mobile node's mobility, and simply use its normal address for communication.

A mobile host is either at home (i.e., located on its **home network**) or it is roaming around the network and located at a **foreign network**. When a mobile host is at home, normal Internet routing ensures that the host can communicate with correspondent hosts.

When the host is roaming, however, its data packets need to be forwarded through a tunnel to the mobile host's current foreign network. This tunneling is done by the **home agent** on the home network and a **foreign agent** on the foreign network.

The basic Mobile IP system therefore consists of three protocols: agent discovery, registration, and tunneling. The mobile host uses the agent-discovery protocol to check regularly whether it is on its home network, on a foreign network, or has moved to another foreign network. When the mobile host notices it has arrived on a (new) foreign network, it requests a care-of address in the foreign network. The host then uses the registration protocol to tell its home agent this care-of address. The home agent then uses the tunnel protocol to send all network packets for the mobile host to the foreign network using the care-of address.

Two extensions to the basic routing protocol are introduced to improve the locality of the basic Mobile IP routing protocols. The first extension is **route optimization**, and aims to keep the network traffic local where possible. The second extension is **regional registrations**, and aims to keep the registration traffic local.

Route optimization avoids the inefficient **triangular routing** that is part of the basic Mobile IP system. Triangular routing refers to the fact that traffic *from* the mobile host goes directly to the correspondent host but that traffic *to* the mobile host must make a detour via the home agent. Route optimization allows a correspondent host to retrieve the current care-of address from the home agent and send its packets directly to the care-of address instead of indirectly via a home agent. The main problems with route optimization are security and backward compatibility. Note that route optimization transforms the Mobile IP system from a routing system into a location-service based system.

With regional registration the concept of a **domain** is introduced. A domain covers several (foreign) networks with associated foreign agents, and has, in turn, also its own foreign agent. The mobile host registers its current care-of address at the foreign agent of the local domain, and the foreign agent of the local domain at the home agent. When the host moves inside the domain, the mobile host needs to update only its binding at the foreign agent of the local domain, and can thus keep the update traffic local. To ensure accessibility of the mobile host, the network traffic is first tunneled to the foreign agent of the local domain and then to the current (normal) foreign agent. Note that this method can, in principle, be extended into a hierarchy of domains and an associated tree of foreign agents.

Evaluation

Mobile IP is designed to support mobility only, and has no support for replication. Its basic architecture is significantly different from the Globe location service because Mobile IP is based on routing. The route optimization extension, however, makes it behave more like a normal location service based on home locations. Mobile IP can easily scale in the number of mobile hosts supported since one can easily increase the number of home agents. Geographical scalability is, however, a problem in the basic design since it does not support locality. Fortunately, the suggested extensions improve the use of locality in the system.

Fault tolerance is dealt with through the use of replication for both home and foreign agents. When a mobile host finds that its home or foreign agent is unresponsive, it can simply select a new one (if available). Security is considered important since the routing structure is vulnerable to denial-of-service attacks. Authentication is therefore an essential part of the registration process.

9.3 Search-tree based Systems

Search-tree based systems, such as our location service, use a distributed search tree to exploit locality in update and lookup operations. The two examples described in this section are search-tree approaches proposed for next-generation mobile-phone systems [Pitoura and Samaras, 2001] and NLS [Hu et al., 2002], which is a variant of the Globe location service.

9.3.1 Next-generation Mobile-phone Systems

Summary

For the next-generation mobile-phone systems, several authors have proposed methods that use a distributed search tree to find user locations, for instance [Wang, 1993]. We use the term “next-generation mobile-phone systems” to refer to a group of similar (partially overlapping) systems that are referred to by various names, such as PCS, PCN, UMTS, and 3G. The infrastructure frequently proposed for user location in these systems is a search tree that represents a distance metric. The methods differ, however, in the way they store location data in the search tree.

In general, the nodes in the distributed search tree store a combination of three types of information: user locations, node pointers, and no information. Different combinations of these types of information have been proposed because they lead to different costs when updating or looking up user locations. For instance, when a user location is stored at a particular node, a lookup operation looking for the user can simply return the location when visiting that node. As such, it might be useful to store a user’s location at all nodes from the leaf node to the root. This is shown in Figure 9.1(a). When a user location is stored at every node, however, every move of the user also results in an update operation that needs to change the location information at every node.

A tree node can also store a pointer to another tree node for a user, indicating that a lookup operation can find more information on the user at the indicated node. For a lookup operation, a stored pointer is less efficient than a user location, but pointers still allow the lookup operation to “home in” on a node that stores the user location. The upside of storing a pointer at a particular node is that a pointer need not always be changed every time the user moves. As long as the user location can still be found via the indicated node, the pointer remains valid and useful. In some proposals pointers can follow only

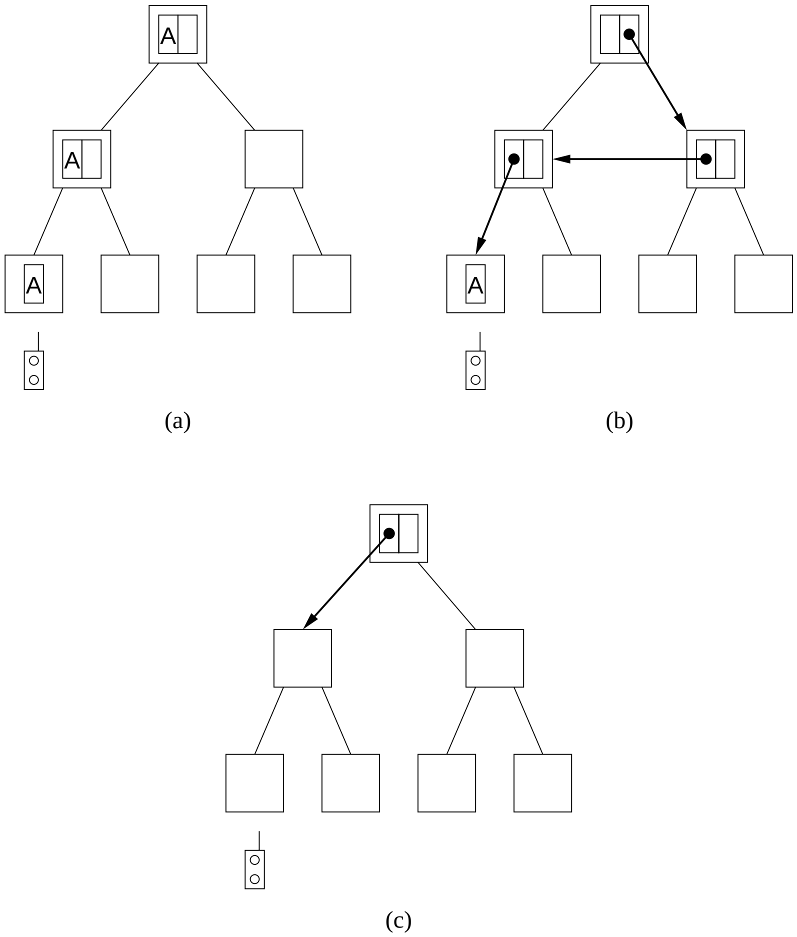


Figure 9.1: Some alternative search tree structures proposed for the next generation mobile phone systems. The user location is depicted by the mobile phone icon. Figure (a) shows a search tree that stores a user location at multiple nodes. Figure (b) shows a search tree where node pointer do not follow the structure of the tree. Figure (c) shows a tree where no user location is stored and brute-force searching is needed.

the structure of the search tree, that is, point to parent or child nodes; in other proposals pointers can point to any node in the search tree, as shown in Figure 9.1(b).

Oddly enough, it can sometimes also be useful to store no information at all in a node for a particular user. When no information is stored, an update operation does not have to change anything. This is particularly attractive when a user is highly mobile, and update traffic would impose too much overhead. The obvious downside of storing no information at a particular node is that a lookup operation can continue only by searching brute force at all neighboring nodes.

The choice of what type of data to store at a particular node can be fixed or be determined dynamically per user. When the type of data stored is determined dynamically, the system frequently looks at the CMR of a user. For instance, when a user is not moving around and is regularly receiving phone calls (i.e., has a high CMR), its location can probably best be stored at many nodes. On the other hand, when a user is frequently moving around and receiving only sporadic phone calls, the search tree can probably best store pointers at a few tree nodes to guide lookup operations in the right direction, and use brute force searching to find the actual user location when needed, as shown in Figure 9.1(c).

It is usually an update operation, caused by a user moving from one location to the next, that inserts or removes an address or pointer at a node. If the search tree system supports caching, however, the lookup operation can also insert or remove addresses and pointers at a node.

Evaluation

Like HLR/VLR systems, next generation mobile-phone systems are designed to support mobility only. There is no concept of replication. When considering the general search tree model, it is clear that the way we store location information in the Globe location service is only a single alternative from a wide variety of alternatives. In this way, the work done in this area is more generic than ours and provides ways to improve the Globe location service. Since the architecture is similar, we can expect the same kind of scalability as for the Globe location service. Unfortunately, most papers in this area focus on the general architecture only, and do not discuss issues such as fault tolerance and security.

9.3.2 NLS

Summary

NLS [Hu et al., 2002] is a combined naming and location service that is designed to support caching and replication in the World Wide Web. NLS was inspired by the Globe project and the Globe location service, and the distributed search tree architecture of NLS is thus similar to ours. A significant functional difference between NLS and the Globe location service is that NLS uses hierarchical (human friendly) names to identify objects instead of (opaque) object handles. NLS therefore has no need for a separate naming service as we do.

NLS makes a number of different choices with regard to storing the name-to-address mapping. First of all, NLS stores the binding of name-to-address always at a leaf node, and intermediate nodes store only forwarding pointers. Furthermore, NLS stores full object names only at leaf nodes. Intermediate nodes store only a hash of the object name together with the forwarding pointer. This decreases the amount of persistent storage needed by NLS. Apart from storing a name-to-address binding, NLS also encourages the storing of more generic names, that is, the prefix of a hierarchical name. For example, the Vrije Universiteit might store besides the name `/nl/vu/www` also the name `/nl/vu`, indicating that all names starting with that prefix can be found at the Vrije Universiteit.

In the area of scalability, NLS uses hash-based partitioning to enable load distribution at the higher-level nodes in the search tree. NLS extends this scheme by also supporting the replication of forwarding pointers at these nodes. To ensure the search tree structure represents distances in the network adequately, NLS uses dynamic spanning trees at lower-level nodes. Unfortunately, the authors do not elaborate on how this works. To improve the efficiency of lookup operations, NLS uses (normal) data caches.

Evaluation

NLS has been mainly built to support replication, but its search tree structure can clearly also support mobility. However, given the fact that normal data caches are used, it is unclear how frequently objects can move without generating an undesirable overhead due to cache misses. We consider the combining of naming and location functionality in a single service a step back since it makes it more difficult to associated multiple human-friendly names with a single object.

The basic architecture of NLS is obviously similar to ours, and similar scalability arguments can thus be made. However, some differences between the designs might proof significant. For instance, the designers of NLS have chosen to use logical partitioning only, and not to improve locality, as we did with our location-based partitioning scheme (see Chapter 3). To improve fault tolerance and availability, NLS uses replication at high-level nodes. The exact consistency protocol used between the replicas, however, is left unspecified, which raises questions on the type of consistency provided and the impact replication has on scalability.

A case is also made for the use of smaller keys since they would allow more names to be stored. However, the computation in Chapter 3 suggests that processing capacity and not storage capacity will probably be the bottleneck in a distributed search tree. Furthermore, providing any form of security would inevitably increase the amount of data stored per object, significantly decreasing the impact of the improvement. Unfortunately, no security information is given on NLS.

9.4 Pointer-chain based Systems

Summary

Pointer-chain systems have been used successfully in local-area systems, and have also been proposed for wide-area systems. Instead of using an object identifier to retrieve the current location of a mobile object, pointer-chain based systems use references to a last known location of a mobile object to communicate with it. To communicate with a mobile object, network packets are basically sent to its last known location (i.e., node). If the object is no longer at that node, the node is responsible for forwarding the packet to the new location of the object. The first work on forwarding pointers was done by Fowler [Fowler, 1985], later work includes Emerald [Jul et al., 1988], Location Independent Invocation (LII) [Black and Artsy, 1990], and SSP Chains [Shapiro et al., 1992].

The core data structure of a pointer-chain system is a list of forwarding pointers maintained by each node in the system. Each entry in the list contains a pointer to the last known location of an object that recently stayed at the node. Every time a local object moves to another node, a new entry is added to this list.

A pointer-chain based system has two important maintenance tasks: *chain reduction* and *garbage collection*. Chain reduction deals with the problem of long paths of forwarding pointers. When a mobile object moves from node to node in a pointer-chain system, it leaves behind a trail of forwarding pointers. When a network packet is sent by a node that stores only a pointer to an old location of the object, the packet will have to follow this trail, resulting in inefficient communication and risking communication failures when a node on the trail has crashed. Nodes in a pointer-chain system therefore always include the most recent location of a mobile object in their communication, allowing the nodes involved in the communication to update their list of forwarding pointers.

Garbage collection deals with the problem of determining when a node in the system can safely remove a forwarding pointer from its list. It is obvious that a node has to keep a forwarding pointer when a local process still uses a remotely located object. Unfortunately, the opposite is not true. When a node has no local processes interested in a remote object, it cannot simply remove its forwarding pointer. The reason for this is that processes at other nodes can still have an old pointer to the current node, expecting it to have a more recent forwarding pointer. Removing the pointer would thus result in breaking the chain of pointers, rendering communication with the referenced object impossible.

There are two ways to deal with the problem of old forwarding pointers. The optimistic approach, used for instance in LII, basically tries to keep all pointers available as long as possible, but allows them to be deleted at any time. In this case, a centralized naming system is used as a fallback mechanism to locate the mobile object if the pointer chain fails. The pessimistic approach, used for instance in SSP chains, basically performs a liveness analysis to determine which forwarding pointers are still in use, and can thus with certainty say which pointers can safely be removed.

Evaluation

Current pointer-chain based system provide support only for mobility and not for replication, even though that might be possible theoretically. An important architectural difference with other naming and location services is that the pointer-chain mechanism is integrated in the communication architecture, that is, there is no separate location service. A security consequence of this is that a user has to trust, in principle, all nodes in the system. Unfortunately, security of pointer-chain systems is usually not discussed. The need for trust in all nodes is also a fault tolerance issue. The main problem of a pointer-chain system is maintaining the connectivity of the pointer chain, which requires either complex algorithms to determine the liveness of forwarding pointers or fallback methods. These complex algorithms and fallback methods, unfortunately, present a scalability problem for pointer-chain based systems.

9.5 Uniform Resource Names

Summary

In this section, we describe the work done by the URN working group of the IETF [URN, 2002]. The goal of the URN working group is to define the Uniform Resource Name (URN) framework, and provide an initial set of components that fit in this framework. Since URNs are persistent identifiers for information resources, they are comparable to the object identifiers used with the Globe location service. The typical example of a URN is an ISBN number that identifies a book.

In contrast to ordinary names, URNs are specifically not required to be human friendly, that is, people do not need to understand the meaning of a URN. URNs should be transcribable by a human, however. The primary use of a URN is to identify a resource. In practice, this means that it can be resolved into the location of a replica of the specified resource, into its properties, or directly into the resource itself.

A fundamental requirement in the URN work is that different types of resources require different types of names. Furthermore, given the desire to support names from existing systems, a large number of legacy resource names are already available. Because of this, the URN framework cannot dictate its own standard syntax and semantics for URNs. Instead, the working group chose to create a global URN name space consisting of several resource-type specific name spaces. Every new type of name that follows the generic URN requirements can simply be added to the global URN name space as a subspace of names.

An important design decision in the URN framework is to make the structure of the name space independent of the structure of servers that perform the name resolution. This way the naming authorities responsible for the content of the URN name space can manage the content independently of the organization performing the URN resolution, allowing multiple organizations to provide name resolution for the same name space.

Another important design decision is the distinction between URN **resolvers** that resolve a specific type of URN and the global **Resolver Discovery Service** (RDS). Since specific URNs are part of type-specific name spaces, they require resolution by specific name resolvers that understand their structure and meaning. It is the task of the RDS to resolve a specific URN into the name or location of a resolver that, in turn, can resolve the URN completely.

The first RDS defined by the working group used DNS to resolve URNs. The proposed RDS is based on rewrite rules that transform the URN into the name of the resolver. The group later generalized this work to include other kinds of names besides URNs, and provided an abstract design, independent of DNS, called the Dynamic Delegation Discovery System (DDDS) to resolve these names.

Evaluation

The URN framework provides a location service for replicated resources only. There is no support for mobility. The main architecture consists of the RDS that guides clients to specific resolvers and resolvers that maintain the exact location of resources. Unfortunately, little information is available on the scalability of the URN approach. We foresee, however, no scalability problems with the RDS itself. The rewrite rules in the RDS are easy to cache and replicate since these rules change only slowly. Local access to these rules should thus frequently be possible.

Unfortunately, no information on the requirements or characteristics of specific resolvers is available. The number of resources that can be supported by specific resolvers is therefore unknown. Furthermore, geographical scalability is mainly a problem of the resolvers since they determine whether or not local replicas of a resource are preferred over nonlocal replicas. The main bottleneck is thus the scalability of the specific resolvers of which little is known.

9.6 Peer-to-peer Systems

In this section we discuss the category of second generation peer-to-peer (P2P) systems, including systems like CAN [Ratnasamy et al., 2001], Chord [Stoica et al., 2001], Pastry [Rowstron and Druschel, 2001], and Tapestry [Zhao et al., 2001]. Peer-to-peer systems provide a *placement service*, that is, a service that decides where in the network the replica's of an object reside if they exists. This is different from a location service, where the object owner is free to choose the locations of the replica's and the location service is only responsible for recording these locations. We use Chord as the example system in this section.

Summary

Chord is described by its designers as “a distributed lookup primitive” that serves as a low-level building block upon which applications can be built, such as the distributed file

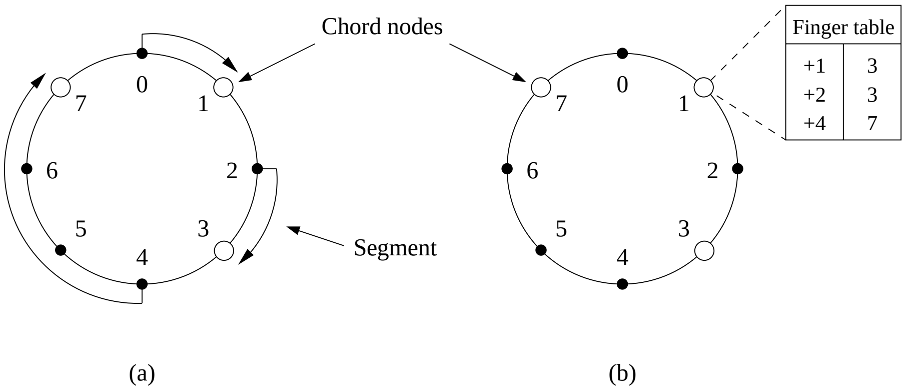


Figure 9.2: The Chord key space with eight keys and three nodes, located at keys 1, 3, and 7. Figure (a) shows the segments the nodes are responsible for. Figure (b) shows the finger table of the node with key 1.

system CFS [Dabek et al., 2001]. In Chord all data can be accessed by a unique key, taken from a large key space. This data is placed at nodes that are responsible for that data. It is the task of Chord to route network packets to the node that is responsible for a piece of data. The exact nature of the node's responsibility for the data and the relation between key and data is determined by the application built on top of Chord, and is irrelevant to Chord system.

To enable routing, Chord divides the key space into segments (ranges), and assigns the responsibility for the keys in these segments to nodes in the system. Basically, each Chord node gets its own unique key, and all data keys are assigned to the node whose key is the smallest node key larger than those data keys. This node key is called the data keys' successor. Figure 9.2(a) shows an example key space with eight keys. This space is divided into three segments that are assigned to the nodes with keys 1, 3, and 7.

To quickly find the node responsible for a segment, Chord stores an index structure, called a finger table, at every Chord node. The finger table is basically a routing table that stores the contact information (e.g., IP address) of Chord nodes with specific keys. The finger table is constructed as follows. Assume that the node that stores the table has key n. The first entry of the table contains the node responsible for key $n+1$, the second entry contains the node responsible for key $n+2$, the third entry contains the node responsible for key $n+4$, etc. In general, the i^{th} entry stores the node responsible for key $n+2^i$. The number of entries in the finger table is the ${}^2log$ of the size of the key space. Note that these values wrap around at the end of the key space.

Figure 9.2(b) shows the finger table of the node 1. Since the key space has eight entries, the finger table has three entries. The first entry of the table stores the node responsible for the key $n+1$, which is node 3. The second entry stores the node for the key $n+2$, which is also node 3. The third and last entry stores the node for the key $n+4$, which is node 7.

Chord routes a network packet for a piece of data to the node that is responsible for that data by recursively forwarding the packet to the node in the finger table whose key is just below the key of the data for which the packet is intended. When there is no node in the finger table with a key just below the data key, the successor of the current node is the node responsible for the data for which the packet is intended. For example, if node 3 in Figure 9.2(b) has a packet for key 0, it will forward the packet to node 7. Node 7 does not have an entry just below key 0, and thus knows its successor (node 1) is responsible for key 0. Since the nodes in a finger table have keys at exponentially increasing distances of the current node, Chord can quickly home in the node responsible for the packet. The nodes in the Chord system also form a ring structure by recording the next and previous node in the key space. This ring allows linear searching as a backup facility for finding nodes when the nodes in the finger table are inaccessible.

Chord has three main tasks: adding new nodes to the system, removing existing nodes from the system, and routing a network packet to a node responsible for a piece of data. Whenever a new node wants to be a part of the Chord system, it first has to determine its own key and take its place in the ring of Chord nodes. Furthermore, the node has to fill its own finger table, and ensure its own presence is know to other nodes to enable them to update their finger tables. Since nodes are expected to regularly join and leave the system, changes to the system have to be made concurrently while still guarding the correctness of the ring structure and finger tables.

The basic routing system selects nodes based only on their nearness in the (logical) key space, and disregards the distances traveled in the network. An extension is proposed, however, to improve the use of locality in Chord. Since the main concern in Chord is that the network packet gets closer (in the key space) to its destination node with every routing step, a Chord node can also forward the packet to a node that makes less progress in the key space but is significantly more nearby in the physical network. By using such nearby nodes, the number of routing steps will increase, but the actual network distance traveled will most likely decrease.

Evaluation

Chord supports replication by making not only the successor node of a key responsible for its data but also other nodes in the vicinity of the key. If the successor node becomes inaccessible, these other nodes can easily take over its responsibility. Clients have no control over the physical locations of replicas, however, since it is the distribution of nodes over the key space that determines which nodes are responsible for a piece of data. Since Chord determines where data should be stored, it has no support for mobility.

To some extent, we can compare the routing in Chord to the location-based and hash-based routing between physical nodes in the Globe location service. In both cases, information in the key (or the Globe object identifier) determines where data will be stored (at which Chord node or which physical location service node). A significant difference is that routing in Chord is solely based on the nodes present in the system, independent of the data stored by Chord nodes, whereas routing in the Globe location service is also guided by the forwarding pointers stored at nodes.

The number of nodes and objects supported by Chord can easily scale since the size of the finger tables in Chord grows only logarithmically. Unfortunately, Chord is inflexible when it comes to the location of an object since this location is determined solely by Chord's placement algorithm and not, for instance, by the object owner. It is thus not possible to place a replica in a hot spot in the (physical) network. Chord can only hope its nodes are distributed evenly, thereby providing local access to data. The routing algorithm itself is able to use nearby nodes while routing packets, but the destination of a packet is always fixed. Geographical scalability can therefore be considered limited.

Fault tolerance in the Chord system is mostly based on replication, that is, making more than one node responsible for a piece of data. The routing functionality of Chord is also highly available since each node can use its own finger table to route packets, independent of other nodes. As with the Globe location service, security is mostly provided by other layers that make use of Chord. It is shown in [Sit and Morris, 2002], however, that Chord nodes can easily determine whether other nodes in Chord can be trusted to route packets correctly. If a node is suspected of routing packets incorrectly, other nodes in Chord can decide to route around this suspicious node.

A significant difference between the Globe location service and Chord, and peer-to-peer systems in general, is in the area of (system) management. Chord is designed to be completely self managing. Unlike (physical) nodes in Globe, nodes in Chord are integrated and removed from the system without any human intervention. This makes Chord attractive for systems without an obvious management organization.

9.7 Summary

In conclusion, we summarize the description of the related work in Table 9.1. For ease of comparison, the first row of the table describes the Globe location service itself. Note that there is no security column since most of the related systems barely describe any security issues. When considering the summary table, the following observations can be made:

- Most systems support either mobility or replication, but not both. Our location service was the first to combine support for both these object characteristics, and thereby inspired NLS to also support both.

- All systems are either lookup or routing based. Six of the systems translate a location-independent name to a location-dependent address. The other three systems route network packets to their proper destination.

- Hierarchical systems are frequently used to deal with scalability. Six of the systems use a hierarchy and two systems do not. The HLR/VLR can be placed in either category. The purpose of the hierarchy, however, differs per system.

Table 9.1: Summary of related work.

System	Support for mobility / replication	Architecture	Scalable in # Objects	Geographical Scalability	Fault Tolerance	Adaptability
Globe Location Service	both	search tree based on a distance metric	yes	yes	partitioned tree nodes	domain and tree changes
DNS	replication	search tree based on name space	yes	yes	replicated zone info	automatic reconfiguration *possible*
HLR/VLR Systems	mobility	two-level structure, HLRs and VLRs	yes	no	N/I	N/I
Mobile IP	mobility	packet forwarding via home location	yes	no	multiple agents	agent discovery
Next-generation Mobile-phone Systems	mobility	search tree based on a distance metric	yes	yes	N/I	N/I
NLS	both	search tree based on a distance metric	yes	yes	replicated & partitioned tree nodes	dynamic spanning trees
Pointer-chain based Systems	mobility	packet forwarding via chain of pointers	yes	no	fallback services	N/A
URNs	replication	two-level structure, RDS and resolvers	RDS: yes, resolvers: N/I	RDS: yes, resolvers: N/I	replicated rewrite rules	N/I
P2P Systems	replication	packet forwarding via hierarchical index	yes	unclear	replicated data & routing info	automatic reconfiguration

N/A:	Not applicable
N/I:	No information available

Chapter 10

Summary and Conclusions

This final chapter concludes this dissertation. It starts with a summary of the preceding chapters, and then discusses some of the general lessons learned through the research on the Globe location service. The chapter ends with a description of the most important open issues, which will need to be addressed by future research.

10.1 Summary

Chapter 1 “Introduction” describes the context and requirements of the research on the Globe location service. The main point the chapter argues is that a location service is needed to support replication and mobility in a distributed system. Furthermore, to operate in a worldwide distributed system, the location service needs to be scalable, both in the number of objects supported and the geographical area covered. Since it is unclear whether existing name services can provide the required location service functionality, a new system is designed from the ground up. The research is structured using a number of research questions that focus on potential problem areas for a scalable location service: locality exploitation, load distribution, efficient use of resources, fault tolerance, security, and maintenance.

Chapter 2 “Architecture” examines the research question: “What architecture combines scalability with flexibility?” The architecture of the location service needs to be flexible to be able to support the methods that deal with the potential problem areas of the location service. The focus in this chapter, however, is mainly on exploiting locality. We solve the locality problem by using a distributed search tree that represents distances in the underlying network. The search tree supports operations to insert, delete, and look up the location(s) of an object. The distributed search tree enables the operations to exploit locality, and thereby supports geographical scalability. Later chapters extend the distributed search tree architecture to deal with the remaining problem areas.

Chapter 3 “Load Distribution” examines the research question: “How do we avoid centralized components in our architecture?” Centralized components should be avoided since they can easily become scalability bottlenecks. The chapter basically deals with two problems, both caused by the root node and other high-level search tree nodes. Note that these nodes are centralized components of the distributed search tree. The first problem examined is the large workload experienced by such high-level nodes. The second problem examined is the long communication distances between such nodes.

We solve both problems by separating the logical structure of the location service (i.e., the way location information is structured), as presented in Chapter 2, from its physical structure (i.e., the way hosts store location information). Using this separation, the large workload problem is easy to solve. By implementing heavily loaded logical nodes using multiple hosts, called physical nodes, we can provide the required storage and processing capacity. We do have to ensure an even load over these physical nodes. The long communication distances problem, however, is more difficult to solve. The goal here is to shorten the network distance traveled by update and lookup operations while they traverse the search tree. We shorten these distances by carefully choosing the physical nodes that store the location information of an object. By choosing physical nodes (one per logical node) that are near each other, we shorten the overall distance traveled by operations.

We examined the efficiency of our load distribution methods using a simulation experiment. The experiment showed that distributing the workload evenly over multiple physical nodes is easily done, for instance, by a hashing scheme. The experiment also showed that by using our distance shortening selection mechanism, we can be shortened the geographical distance traveled by an operation by about 20%. It is, unfortunately, unclear how much effect that 20% has on the actual network distance traveled. Further research is therefore needed to examine the long communication distance problem, for instance, by trying other physical node selection methods.

The research described in Chapter 4 “An Efficient Lookup Operation” concerns improving the performance of the lookup operation. The chapter examines the problem of performing the same tree traversal each time when looking for the same object multiple times. If we can avoid performing the same tree traversal multiple times, for instance, by caching, we can improve the scalability of the location service by using fewer nodes in the search tree, especially high-level nodes. Unfortunately, ordinary caching techniques do not work well with highly mobile objects since these objects change their location too frequently to be reused by an ordinary caching scheme.

We solve the mobility problem by recognizing that even a mobile object has a stable “location,” namely its mobility domain. By storing the location of an object in the tree node associated with the mobility domain of that object, the location service can cache a reference to that tree node in other tree nodes, and short-cut the tree traversal by directing the lookup operation directly to the referenced node. We investigated the effectiveness of our caching method using a simulation experiment. The experiment showed that the reference caching method is superior to normal caching and no caching. It can reduce the overall workload by more than 30% compared to a search tree without caching and reduce the distance traveled by lookup operations by almost 50%.

Chapter 5 “Availability and Fault Tolerance” examines the research question: “How do we ensure availability in a huge system such as our location service?" Providing high availability is important because the location service plays a central role in communication. Since the location service is a huge system, however, it is very likely that partial failures will occur. To maintain high availability, these partial failures should have only a minimal effect on the location service as a whole. The main problem we have to deal with is that partial failures might result in inconsistencies in the distributed state of the location service. Since these inconsistencies limit the availability of the location information stored in the location service, a fault tolerance mechanism is needed to resolve the inconsistencies and keep the information available.

The main point argued in this chapter is that only a few small changes are needed to implement fault tolerance in the location service and improve its availability. Since these changes are only small, the location service remains comprehensible. We basically solve the (in)consistency problem by using and strengthening the inherent characteristics of the update operations, such as their idempotency and atomicity. Given these characteristics and the structure of the traversal of update operations in the search tree, consistency is easily restored. The main result of this chapter is that fault tolerance can be implemented in the Globe location service in a simple and lightweight fashion.

Chapter 6 "Security" examines the research question: "What kind of security is needed for the information stored in the location service, and how do we provide it?" Since the information stored in the location service is not confidential and clients check the identity of the objects they use, the main security problem is denial-of-service (DoS) attacks. The location service is a prime target for people that want to disrupt the communication in a distributed system since the service is pivotal to all communication. The main point made in this chapter is that we can use well-known security mechanisms to provide the necessary security guarantees. Since we can use these simple, well-known techniques, we can have high confidence that the location service is well protected.

The location service deals with DoS attacks by providing access control on update operations and integrity control on communication channels. To provide access control on the stored location information, the location service uses certificates that give users specific update rights. The location service uses the normal combination of public-key and shared-key cryptography to provide communication integrity. Other protection techniques, such as puzzle solving, can be added when the need arises. The efficiency of the security mechanisms, however, has not yet been examined.

Chapter 7 "Tree Management" examines the research question: "How do we ensure that the location service can deal with changes in its environment?" To keep the location service efficient and effective, the logical and physical structure of its distributed search tree needs to be regularly adjusted. The logical structure needs to be adjusted to match changes in the distances in the underlying network, and the physical structure needs to be adjusted to match changes in the workload. The main goal of the research in this chapter is to show the feasibility of tree management.

The problem examined in this chapter is how to perform these changes while keeping the location service operational. This problem can be divided into two subproblems: performing logical changes and performing physical changes. We solved both subproblems by identifying a number of simple, basic changes, showing how these changes can be performed, and showing how to combine them into more complex changes. Since the main goal was a feasibility study, actual algorithms are not provided and their efficiency is thus also not examined.

The research in Chapter 8 “Prototype Implementation” examines the performance of an actual implementation of the Globe location service. We built a prototype implementation of the Globe location service to show that an actual implementation does not limit the scalability of the location service design, as presented in Chapters 2–7. To simplify the performance measurements, we focused on the performance of individual physical nodes.

The measurements showed that an individual physical node can perform 9 update operations or 107 lookup operations per second. Using these performance numbers and some general assumptions, we estimated the performance of the Globe location service as a whole. These estimates showed, for instance, that in a search tree with four layers an update operation will last between 300 and 600 ms and that a lookup operation will last between 15 and 650 ms. These estimates support our claim that a scalable location service can be built. Furthermore, the profiling information shows that an optimized version can provide even better results.

Chapter 9 “Related Work” contains a comparison of the Globe location service with similar systems. The chapter discusses example systems from each of the following categories of related work: naming and directory services, home-location based systems, search-tree based systems, pointer-chain based systems, uniform resource names (URNs), and peer-to-peer (P2P) systems. We compare the example systems to the Globe location service on the points raised in Chapters 2–7: replication support, mobility support, basic architecture, scalability, fault tolerance, security, and management (adaptability). The chapter shows that the location service is different both in its goals (i.e., its support for replication as well as mobility) and in its heavy scalability requirements (i.e., the number of objects supported and the area covered by the service).

10.2 Lessons Learned

The main research question examined in this dissertation is: ”How can we build a worldwide location service?” We answered this research question by providing both the design of the Globe location service and an evaluation of its viability. The design of the Globe location service can be characterized using the following features:

- A distributed and partitioned search tree that represents distances in the underlying network, as explained in Chapter 2 and Chapter 3.
- An RPC-based communication system with at-least-once failure semantics and idempotent remote procedures, as explained in Chapter 5.
- A caching technique based on references to other nodes in the search tree, as explained in Chapter 4.

Table 10.1: Main requirements of the Globe location service.

Support for replication
Support for mobility
Support for 10^{12} objects
Support for objects distributed worldwide
Support for exploiting locality

- A certificate based access control mechanism to prevent unauthorized modifications to location information, as explained in Chapter 6.
- A method to modify the logical and physical structure of the search tree while the service remains operational, as explained in Chapter 7.

The main requirements of the Globe location service, which were described in Chapter 1, are summarized in Table 10.1. We base our claim that the design of the location service meets these requirements on the following three arguments:

- The design has been thoroughly analyzed with respect to its requirements.
- The behavior of the Globe location service has been simulated to examine its scalability and performance.
- The performance of a prototype implementation of the Globe location service has been measured.

Besides giving the answer to the main research question, we also make the following, more general observations:

O1 *The performance of the Globe location service is mainly determined by the performance of its I/O hardware.*

The performance of the location service is mainly determined by the time spent using hard disks and on the network, and not by the time spent executing algorithms and protocols. This strong influence of the I/O hardware is present both in the throughput of the location service and the duration of its operations.

The throughput of the location service is mainly determined by the throughput of its physical nodes. The network plays no role because its throughput (i.e., bandwidth) is much larger than the throughput of the physical nodes. Since Chapter 8 has shown that hard disk accesses account for 60%–95% of the time spent in physical nodes, hard disk performance clearly has a significant impact on the throughput of the location service as a whole.

The duration of individual operations, on the other hand, is determined both by the time spent in physical nodes and the time spent on the network. Since hard disk accesses mainly determine the time spent in a node and wide-area network latency is in the same order of magnitude as hard disk accesses (10 ms or more), their combined influence on the duration of update and lookup operations is significant.

While we have been able to mask the influence of I/O hardware on the throughput of the Globe location service using node partitioning, we have, unfortunately, not been able to mask the influence of I/O hardware on the duration of operations.

O2 *Separating the logical and physical design of the Globe location service has proven useful.*
The design of the Globe location service shows a distinct separation between its logical and physical design. We gained the most of the scalability of our design by this separation, which allowed us to focus on locality with the logical design and on workload with the physical design. If we would have considered an integrated logical and physical design, it is likely that we would have dismissed a design based on the distributed search tree solution early on. We consider this separation therefore a powerful feature of our design.

O3 *The highly modular design of the Globe location service has made its prototype easy to implement.*
The Globe location service has a modular design, with each module or layer adding its own functionality to the system as a whole. This design follows the "separation of concerns" strategy, which is used for managing the (potential) complexity of building large applications. The modular design has minimal dependencies between its layers and modules, allowing us to understand each module and layer by itself. The Globe location service therefore has a comprehensible design. Such a comprehensible, modular design provides a solid basis for further development.

10.3 Future Work

We can divide future work into short-term, medium-term, and long-term goals. The short-term goals have a strong implementation focus, while the long-term goals have a strong research focus. The medium-term goals have a focus that is both implementation and research oriented.

Short Term Goals

Since the prototype described in Chapter 8 is incomplete, the obvious next step in our research is to complete the location service prototype by adding the remaining functionality described in this dissertation. The remaining functionality consists of the move-up and move-down operations from Chapter 4 and all the security features from Chapter 6. A complete prototype will allow us to make more realistic performance measurements, which, in turn, will enable a more detailed analysis of the performance and scalability of the Globe location service. Furthermore, a complete prototype will allow us to implement client applications, which, in turn, allows us to study the everyday use of the location service by real applications.

After completing the prototype, we can look for new ways to improve its performance. The main method to improve the performance of the prototype is by introducing more concurrency. More concurrency is needed to mask the high latency of hard disk accesses, as shown by Chapter 8. The easiest way to introduce more concurrency is by implementing asynchronous database operations. With these asynchronous operations, the prototype as a whole does not have to wait for each, single database operation to finish, but instead can continue doing other work in parallel. Note that this method is analogous to the concurrent RPC mechanism described in Chapter 5.

A more general solution to the concurrency problem is to provide a fully multi-threaded implementation. A fully multi-threaded version would, for instance, also allow the prototype to make use of multi-processor hardware. Unfortunately, a fully multi-threaded version does require a major redesign of the current prototype, and would thus require a significant amount of work. Either way, an implementation with improved concurrency will allow us to make more realistic predictions of the scalability of the location service.

Medium Term Goals

A specific part of the Globe location service research that was left open in Chapter 7 are the details of the tree management methods, such as data structures, algorithms, and protocols. When these details are integrated in the location service prototype, the prototype will allow us to examine the amount of effort required to keep the location service operational, especially when it has become a large service. If it proves difficult to keep a large location service operational, this difficulty could severely limit the scalability of the location service. The research should be straightforward since it basically consists of filling in the missing details. A logical first step would be to focus on partitioning changes since they are more straightforward than search tree changes.

Long Term Goals

A more general issue to look at is the characteristics of realistic mobility, replication, and lookup patterns and how they affect the performance of the location service. These characteristics are interesting because we expect the performance (and thus scalability) of the location service to depend heavily on the type of use. For instance, if there is much locality in the patterns, the location service will be very scalable. Using these patterns, we can thus validate the design decision made in Chapter 2 and Chapter 4. We can find these real-life mobility, replication, and lookup patterns by examining the client applications, which were made possible by our complete, real-life location service prototype.

A part of the research on the Globe location service that should be reexamined are the locality characteristics of the load distribution schemes of Chapter 3. While it is possible to provide an even load over the physical nodes, exploiting locality using load distribution has proven to be more difficult. We should therefore further examine the effects of our (geographical) location-aware selection method on the network distances traveled by update and lookup operations, and see if we can devise other methods that provide better results.

Finally, we also need to consider the operational side of the location service. The main research question to be answered is "Can a single (virtual) organization manage the large number (i.e., $O(10^5)$) of physical nodes, spread all over the world?" This management problem becomes even more difficult, if we consider an underlying network with frequently changing distances. Recall that management by a single (virtual) organization is an assumption of Chapter 6 and Chapter 7. If changes are needed in the structure of the Globe location service support organization, these changes will undoubtedly have effects on the security mechanisms used in the location service and the security guarantees provided. If, as a consequence, changes are needed in the security design, these changes could, in turn, adversely affect the performance and scalability of the Globe location service.

Bibliography

Akamai Technologies, Inc. (2002). EdgeSuite. `http://www.akamai.com/`.

Albitz, P. and Liu, C. (1992). *DNS and BIND*. O'Reilly & Associates, Sebastopol, CA, USA.

Ateniese, G. and Mangard, S. (2001). A New Approach to DNS Security (DNSSEC). In *Proc. of 8th ACM Conference on Computer and Communications Security (CCS-8)*, Philadelphia, PA, USA. ACM.

Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. (1992). Non-volatile memory for fast, reliable file systems. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*.

Bakker, A. (2002). *An Object-based Software Distribution Network*. PhD thesis, Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands.

Ballintijn, G., van Steen, M., and Tanenbaum, A. (2000). Characterizing Internet Performance to Support Wide-area Application Development. *Operating Systems Review*, 34(4):41–47.

Ballintijn, M. K. (1994). *The ratio of structure functions for the neutron and the proton*. PhD thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands.

Ballintijn, M. R. (1999). *Vocal variation in the Collared dove : coding messages in coo-vocalizations*. PhD thesis, Universiteit Leiden, Leiden, The Netherlands.

Barak, A. and Litman, A. (1985). MOS: a multicomputer distributed operating system. *Software — Practice and Experience*, 15(8):725–737.

Birrell, A. and Nelson, B. (1984). Implementing Remote Procedure Calls. *ACM Trans. Comp. Syst.*, 2(1):39–59.

Black, A. P. and Artsy, Y. (1990). Implementing Location Independent Invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119.

Butterfield, D. A. and Popek, G. J. (1984). Network Tasking in the Locus Distributed Unix System. In *Proc. of the USENIX Summer Conference*, pages 62–71.

Cate, V. (1992). Alex - A Global Filesystem. In *Proc. Usenix File Systems Workshop*, pages 1–11, Ann Harbor, MI, USA. USENIX.

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. (1994). RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185.

Cheriton, D. and Mann, T. (1989). Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. Comp. Syst.*, 7(2):147–183.

Comer, D., Droms, R. E., and Murtagh, T. P. (1990). An Experimental Implementation of the Tilde Naming System. *USENIX Computing Systems*, 3(4):487–515.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*, Chateau Lake Louise, Banff, Canada.

Diffie, W. and Hellman, M. E. (1976). New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654.

Duvvuri, V., Shenoy, P., and Tewari, R. (2000). Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proc. of IEEE INFOCOM 2000*, pages 834–843, Tel Aviv, Israel. IEEE.

Fowler, R. J. (1985). Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Dept. of Computer Science, Univ. of Washington, Seattle, WA, USA.

Gray, C. and Cheriton, D. (1989). Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th ACM Symp. Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, USA. ACM.

Harrison, C. G., Chess, D. M., and Kershenbaum, A. (1995). Mobile Agents: Are They a Good Idea. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY.

Hoare, C. (1974). Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557.

Homburg, P. (2001). *The Architecture of a Worldwide Distributed System*. PhD thesis, Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands.

Hu, Y. C., Rodney, D. A., and Druschel, P. (2002). Design and Scalability of NLS, a Scalable Naming and Location Service. In *Proc. of IEEE Infocom 2002*, New York, NY, USA.

Huck, P., Butler, M., Gupta, A., and Feng, M. (2002). A self-configuring and self-administering name system with dynamic address assignment. *ACM Transactions on Internet Technology (TOIT)*, 2(1):14–46.

Internet Software Consortium (2002). Internet Domain Survey, Jan 2002. `http://www.isc.org/ds/WWW-200201/index.html`.

Jannink, J., Lam, D., Shivakumar, N., Widom, J., and Cox, D. (1997). Efficient and Flexible Location Management Techniques for Wireless Communication Systems. *Journal of Wireless Networks*, 3(5):361–374.

Johnson, D. B. and Zwaenepoel, W. (1987). Sender-Based Message Logging. In *Proc. of the 17th Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19, Pittsburgh, PA, USA. IEEE.

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

Jung, J., Sit, E., Balakrishnan, H., and Morris, R. (2001). DNS Performance and the Effectiveness of Caching. In *Proc. of the ACM SIGCOMM Internet Measurement Workshop*.

Kopetz, H. and Verissimo, P. (1993). Real Time and Dependability Concepts. In Mullender, S., editor, *Distributed Systems*, pages 411–446. Addison-Wesley, 2 edition.

Kuz, I. (2003). *An Approach to Scalable Wide-Area Web Servers*. PhD thesis, Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands.

Lampson, B. W. (1986). Designing a Global Name Service. In *Proc. 5th ACM Symposium on Principles Of Distributed Computing*, pages 1–10, Calgary, Alberta, Canada. ACM. Presented the previous year, ISBN 0-89791-198-9.

Lampson, B. W. and Sturgis, H. E. (1979). Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center.

Laprie, J.-C. (1995). Dependability – Its Attributes, Impairments and Means. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewood, B., editors, *Predictably Dependable Computing Systems*, pages 3–24. Springer-Verlag, Berlin, Germany.

Lee, E. K. and Thekkath, C. (1996). Petal: Distributed Virtual Disks. In *Proc. of the ACM 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, USA. ACM.

Leighton, F. and Lewin, D. (2000). Global Hosting System. United States Parent 6,108,703.

Leiwo, J., Aura, T., and Nikander, P. (2000). Towards network denial of service resistant protocols. In *Proc. of the 15th International Information Security Conference (IFIP/SEC 2000)*, Beijing, China. Kluwer.

Loshin, P., editor (2000). *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. Morgan Kaufmann, San Francisco, CA, USA.

Mazières, D. and Kaashoek, M. F. (1998). Escaping the evils of centralized control with self-certifying pathnames. In *Proc. of the 8th ACM SIGOPS European Workshop*.

Milne, R. E. and Strachey, C. (1976). *A Theory of Programming Language Semantics*. Chapman and Hall.

Mobile IP (2002). Mobile IP Working Group of IETF. http://www.ietf.org/html.charters/mobileip-charter.html.

Mockapetris, P. (1987). RFC 1034: Domain Names - Concepts and Facilities.

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

National Institute of Standards (1995). Secure Hash Standard. Technical Report FIPS-180-1, U.S. Department of Commerce. Also known as: 59 Fed Reg 35317 (1994).

Panzieri, F. and Shrivastava, S. (1988). Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing. *IEEE Trans. on Software Engineering*, 14(1):30–37.

Perkins, C. (2002). RFC 3220: IP Mobility Support for IPv4.

Perkins, C. E. (1998). Mobile Networking with Mobile IP. *IEEE Internet Computing*, 2(1):58–69.

Pitoura, E. and Samaras, G. (2001). Locating Objects in Mobile Computing. *IEEE Transactions on Kowledge and Data Engineering*, 13(4):571–592.

Radicati, S. (1994). *X.500 Directory Service: Technology and Deployment*. International Thomson Computer Press, London.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM 2001*, pages 161–172, San Diego, CA, USA. ACM.

Rhind, D. (1991). Cartographical-related research at Birbeck College 1987-91. *The Cartographic Journal*, 28:63–66.

Rivest, R., Shamir, A., and Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commununications of the ACM*, 21(2):120–126.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany.

Schuba, C. L., Krsul, I. V., Kuhn, M. G., Spafford, E. H., Sundaram, A., and Zamboni, D. (1997). Analysis of a Denial of Service Attack on TCP. In *Proc. of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society, IEEE Computer Society Press.

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

Sit, E. and Morris, R. T. (2002). Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Proc. of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA.

Spector, A. Z. (1982). Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):246–260.

Steiner, J. G., Neuman, C., and Schiller, J. I. (1988). Kerberos: An Authentication Service for Open Network Systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, USA.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM '01 Conference*, San Diego, CA, USA.

Stoker, G., White, B. S., Stackpole, E., Highley, T., and Humphrey, M. (2001). Toward Realizable Restricted Delegation in Computational Grids. In *High Performance Computing and Networking (HPCN 2001), Proc. of European 9th International Conference*, pages 32–41, Amsterdam, The Netherlands.

URN (2002). URN Working Group of IETF. `http://www.ietf.org/html.charters/urn-charter.html`.

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

Wieringa, R. and de Jonge, W. (1995). Object Identifiers, Keys, and Surrogates - Object Identifiers Revisited. *Theory and Practice of Object Systems*, 1(2):101–114.

Zhao, B. Y., Kubiatowicz, J., and Joseph, A. D. (2001). Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U.C. Berkeley.

Index

Samenvatting

Het Vinden van Objecten in een Wereldwijd Systeem

Het huidige Internet biedt zijn gebruikers een grote hoeveelheid diensten aan die, bijvoorbeeld, informatie verschaffen. Het bekendste onderdeel van het Internet dat informatie verschaft is waarschijnlijk het World Wide Web (WWW), waar gebruikers informatie kunnen vinden over onderwerpen zo divers als wereldpolitiek en aspergerecepten. Daarnaast heeft het WWW ook het ontstaan van e-commerce mogelijk gemaakt. Een andere belangrijke dienst die door het Internet geleverd wordt is communicatie tussen Internet gebruikers. De bekendste vormen hiervan zijn waarschijnlijk e-mail en instant messaging.

De laatste tien jaar is het Internet explosief gegroeid. Deze groei is zichtbaar zowel in het aantal gebruikers van het Internet als in de grote hoeveelheid en verscheidenheid aan diensten die op het Internet worden aangeboden. Daarnaast is het Internet gegroeid van een netwerk dat alleen de Verenigde Staten van Amerika bedekte tot een netwerk dat alle uithoeken van de wereld met elkaar verbindt.

Helaas heeft de explosieve groei van het Internet ook schaalbaarheidsproblemen geïntroduceerd. Met de grote hoeveelheid gebruikers in het Internet lopen de computers die diensten verlenen het gevaar overbelast te raken. Ze kunnen simpelweg niet de grote hoeveelheid verzoeken voor informatie verwerken. Daarnaast geldt ook dat, hoewel het mogelijk is om met het Internet informatie van de andere kant van de wereld te halen, dit vaak niet wenselijk is gegeven de inherente traagheid van communicatie over deze afstanden.

Om deze schaalbaarheidsproblemen op te lossen zijn de verschaffers van Internet diensten begonnen met het repliceren van diensten. Replicatie refereert aan het verschaffen van dezelfde dienst op meerdere computers op verschillende plaatsen in het Internet. Replicatie stelt verschillende gebruikers in staat verschillende replica's van een Internet dienst te gebruiken, waardoor de totale belasting van de dienst verspreid wordt over meerdere computers. Daarnaast stelt het verspreiden van replica's over het Internet gebruikers in staat een dichtbij gelegen replica van een dienst te gebruiken, waardoor de traagheid van langeafstandscommunicatie kan worden vermeden en de belasting van het netwerk kan worden beperkt.

Mobiel computer gebruik is een andere belangrijke trend in de afgelopen tien jaar, die mogelijk werd gemaakt door de introductie van draagbare computers, zoals laptops en palmtops. Computers staan niet langer stil op een vaste plaats met een vaste verbinding aan het Internet, maar vergezellen gebruikers tijdens hun reizen en stellen deze gebruikers in staat computer functionaliteit overal en altijd te gebruiken, zelfs wanneer er gebruik moet worden gemaakt van het Internet.

Om met een dienst in het Internet te communiceren moet een gebruiker weten wat de lokatie van de computer is die die dienst verzorgt. Echter, aangezien de computers die diensten leveren mobiel kunnen zijn, kunnen we niet zeker zijn dat een dienst beschikbaar blijft op dezelfde lokatie. Daarnaast kunnen we ook niet zeker zijn dat de huidige dichtstbijzijnde replica van een dienst ook in de toekomst de dichtstbijzijnde replica blijft. Wat we dus nodig hebben is een manier om de huidige lokaties van de replica's van een dienst bij te houden.

Het bijhouden van de lokaties van Internet diensten wordt traditioneel gedaan door naamgevingsdiensten, zoals bijvoorbeeld DNS. Naamgevingsdiensten kunnen allerlei soorten informatie opslaan voor een benoemde dienst, inclusief zijn lokatie(s). Door gebruikers te laten refereren naar een dienst via zijn naam, in plaats van via zijn lokatie, kunnen naamgevingsdiensten gebruikers afschermen van problemen zoals waar een dienst zich op dit moment bevindt, of een dienst uit meerdere replica's bestaat, en of een dienst van lokatie kan veranderen. Met andere woorden, naamgevingsdiensten leveren respectievelijk lokatie onafhankelijkheid, replicatie onafhankelijkheid, en migratie onafhankelijkheid.

De naamgevingsdiensten die heden ten dage gebruikt worden zijn echter niet in staat om de lokaties van grote hoeveelheden gerepliceerde en mobiele diensten op te slaan en te verwerken. Als we dus meer Internet toepassingen in staat willen stellen gebruik te maken replicatie en mobiliteit, hebben we een nieuw soort naamgevingsdienst nodig die specifiek ontworpen is voor het bijhouden van de lokatie(s) van grote hoeveelheden diensten op een wereldwijde schaal. We noemen deze nieuwe specifieke naamgevingsdienst een lokalisatiedienst.

De centrale onderzoeksvraag in dit proefschrift is hoe we een wereldwijde lokalisatiedienst kunnen bouwen. Om deze centrale onderzoeksvraag te beantwoorden hebben we hem opgedeeld in een aantal specifieke onderzoeksvragen, die we per hoofdstuk beantwoorden. Deze specifieke onderzoeksvragen kijken naar potentiële probleemgebieden voor een lokalisatiedienst, zoals het gebruiken van lokaliteit, het verspreiden van de werkdruk, het efficiënt gebruik maken van hardware, het verschaffen van fout tolerantie en veiligheid, en het doen van onderhoud. De oplossing voor het lokalisatievraagstuk die we in dit proefschrift voorstellen is de Globe lokalisatiedienst.

Hoofdstuk 2 "Architecture" behandelt de onderzoeksvraag: "Wat voor architectuur combineert schaalbaarheid met flexibiliteit?" De architectuur van de lokalisatiedienst moet flexibel zijn om de oplossingen voor de verschillende probleemgebieden van de lokalisatiedienst te ondersteunen. De focus in dit hoofdstuk ligt daarnaast op het gebruik van lokaliteit, met andere woorden het vinden van dichtbijgelegen replica's. We hebben het lokaliteitsprobleem opgelost met behulp van een gedistribueerde zoekboom die afstanden representeert in het onderliggende netwerk. Deze zoekboom biedt operaties aan om de lokatie(s) van de replica's van een Internet dienst toe te voegen, te verwijderen, en op te zoeken.

Hoofdstuk 3 "Load Distribution" behandelt de onderzoeksvraag: "Hoe gaan we om met centrale componenten in onze architectuur?" Centrale componenten vormen een risico omdat deze gemakkelijk tot schaalbaarheidsproblemen kunnen leiden. Helaas bevat het ontwerp, zoals besproken in Hoofdstuk 2, dit soort centrale componenten, namelijk de knopen die hoog in de zoekboom liggen. Deze knopen veroorzaken twee problemen. Het eerste probleem betreft de grote werkdruk van zulke knopen. Het tweede probleem betreft de grote afstand tussen zulke knopen, en de daaruitvolgende communicatie traagheid.

We kunnen beide problemen oplossen door onderscheid te maken tussen de logische structuur en de fysieke structuur van de lokalisatiedienst. De logische structuur betreft de manier waarop (lokatie) informatie in de zoekboom is gestructureerd, en de fysieke structuur betreft de manier waarop computers deze informatie feitelijk opslaan. Met dit onderscheid is het simpel om het probleem van overbelasting op te lossen. Door zwaar belaste logische knopen te implementeren met meerdere computers, genaamd fysieke knopen, kunnen we de vereiste opslag- en verwerkingscapaciteit verschaffen. We moeten er dan wel voor zorgen dat de computers gelijkmatig belast worden. Het probleem van communicatie over lange afstanden is moeilijker op te lossen. Het doel is hier om de afstanden die door opzoek- en veranderingsoperaties worden afgelegd zo kort mogelijk te houden. We proberen deze afstanden kort te houden door de lokaties van Internet diensten op te slaan in fysieke knopen die dicht bij elkaar en bij gebruikers liggen.

We hebben de effectiviteit van onze techniek voor het verspreiden van de belasting en het verkorten van communicatie afstanden onderzocht met behulp van een simulatie experiment. Dit experiment liet zien dat het gelijkmatig verspreiden van de belasting gemakkelijk te doen is. Het experiment liet ook zien dat onze methode voor het verkorten van afstanden tot een reductie van 20% van de afgelegde weg leidt. Meer onderzoek is echter nodig om te bepalen wat het effect van deze 20% is op de lokalisatiedienst als geheel.

Het onderzoek beschreven in Hoofdstuk 4 "An Efficient Lookup Operation" betreft het efficiënter maken van de opzoekoperatie. Het hoofdstuk onderzoekt het probleem dat als we de lokatie van dezelfde Internet dienst meerdere keren opzoeken de zoekboom elke keer opnieuw op dezelfde manier doorlopen wordt. Als we het meerdere keren doorlopen van de zoekboom kunnen voorkomen, bijvoorbeeld met het gebruik van een caching techniek, dan wordt de schaalbaarheid van de lokalisatiedienst verbeterd en heeft de lokalisatiedienst minder fysieke knopen nodig. Normale caching technieken werken echter niet wanneer zeer mobiele diensten ondersteund moeten worden omdat de lokaties van dit soort diensten te snel veranderen om hergebruikt te worden. Een ander soort caching techniek moet dus bedacht worden.

We hebben het mobiliteitsprobleem opgelost met het inzicht dat zelfs een mobiele dienst een stabiele "lokatie" heeft, namelijk zijn mobiliteitsdomein. Met de term mobiliteitsdomein refereren we naar het domein waarbinnen een dienst zich beweegt. Door de lokatie van een Internet dienst op te slaan in de knoop (in de zoekboom) die behoort bij het mobiliteitsdomein van die dienst kan de lokalisatiedienst *stabiele* verwijzingen opslaan naar die knoop. De lokalisatiedienst kan deze stabiele verwijzingen vervolgens gebruiken tijdens het opzoeken van lokaties om het doorlopen van de boom in te korten door direct naar de aangeweze knoop te springen.

We hebben de effectiviteit van onze verwijzing caching techniek onderzocht met behulp van een simulatie experiment. Dit experiment liet zien dat onze caching techniek de efficiëntie van de lokalisatiedienst verbeterde. Onze verwijzing caching techniek verminderde de totale belasting van het systeem met 30% vergeleken met een systeem zonder caching, en het verminderde de totaal afgelegde weg van een opzoekoperatie met bijna 50%.

Hoofdstuk 5 “Availability and Fault Tolerance” behandelt de onderzoeksvraag: “Hoe kunnen we de continue beschikbaarheid garanderen van een groot systeem zoals de lokalisatiedienst?” Het garanderen van de continue beschikbaarheid is belangrijk omdat de lokalisatiedienst een centrale rol speelt in alle communicatie tussen gebruikers en diensten.

Aangezien de lokalisatiedienst een groot systeem is, is het echter zeer waarschijnlijk dat fouten zullen optreden in het systeem. Om continue beschikbaarheid te garanderen mogen deze fouten dus slechts een minimale invloed hebben op de werking van het systeem als geheel. Het belangrijkste probleem is dat fouten kunnen leiden tot inconsistenties in de informatie die ligt opgeslagen in de zoekboom, en deze inconsistenties kunnen, op hun beurt, de beschikbaarheid van informatie, zoals lokaties, beperken. Om deze beperkte beschikbaarheid te voorkomen hebben we mechanismes nodig die dit soort inconsistenties verbeteren.

Dit hoofdstuk laat zien dat slechts een aantal kleine veranderingen nodig is om de inconsistenties te verbeteren en de beschikbaarheid van lokaties te garanderen. Aangezien deze veranderingen slechts klein zijn, blijft de lokalisatiedienst als geheel begrijpelijk. We lossen het inconsistentieprobleem op door het gebruik maken en het versterken van de inherente eigenschappen van de operaties die informatie veranderen, zoals idempotentie en atomiciteit. Het belangrijkste resultaat van dit hoofdstuk is dat fouttolerantie gemakkelijk toegevoegd kan worden aan de lokalisatiedienst, en geen schaalbaarheidsprobleem vormt.

Hoofdstuk 6 “Security” behandelt de onderzoeksvraag: “Wat voor veiligheidsgaranties zijn nodig voor de lokalisatiedienst en hoe kunnen we die verschaffen?” Aangezien de lokaties die opgeslagen liggen in de lokalisatiedienst niet geheim zijn en gebruikers zelf controleren of ze met de correcte dienst communiceren, is het belangrijkste veiligheidsprobleem een zogenaamde *denial-of-service* (DoS) aanval, die legitieme gebruikers de toegang tot een Internet dienst ontzegt. De lokalisatiedienst is een belangrijk doelwit voor mensen die de communicatie in een computernetwerk willen verstoren omdat de lokalisatiedienst een centrale rol speelt in alle communicatie. Dit hoofdstuk laat zien dat de vereiste veiligheidsgaranties geleverd kunnen worden met het gebruik van simpele en bekende beveiligingstechnieken. Vanwege het gebruik van deze simpele en bekende technieken kunnen we een groot vertrouwen hebben dat de lokalisatiedienst goed beveiligd is.

De lokalisatiedienst verweert zich tegen DoS aanvallen door gebruik te maken van een combinatie van technieken voor toegangscontrole en communicatieintegriteit. Toegangscontrole wordt verschaft door gebruik te maken van cryptografische certificaten die gebruikers specifieke rechten geven om lokaties van een dienst toe te voegen of te verwijderen. De lokalisatiedienst gebruikt verder een combinatie van *public-key* en *shared-key* cryptografie voor het verkrijgen van communicatie integriteit. Andere beschermingstechnieken, zoals bijvoorbeeld cryptografische puzzels, kunnen toegevoegd worden als dat nodig blijkt.

Hoofdstuk 7 “Tree Management” behandelt de onderzoeksvraag: “Hoe kunnen we er voor zorgen dat de lokalisatiedienst om kan gaan met veranderingen in zijn omgeving?” Om de lokalisatiedienst efficiënt en effectief te houden moet zowel de logische als de fysieke structuur van de zoekboom regelmatig aangepast worden aan de huidige toestand van het onderliggende netwerk. De logische structuur moet worden aangepast om veranderingen in de afstanden in het netwerk te volgen, en de fysieke structuur moet worden aangepast om de verdeling van de werkdruk over fysieke knopen gelijkmatig te houden. Omdat het beheren van de zoekboom een groot onderzoeksgebied is, hebben we het doel van het onderzoek in dit hoofdstuk beperkt tot het aantonen van de haalbaarheid van het beheren van de logische en fysieke structuur van de zoekboom.

Het probleem dat we onderzoeken in dit hoofdstuk is hoe veranderingen in de zoekboom doorgevoerd kunnen worden terwijl de lokalisatiedienst operationeel blijft. Deze veranderingen kunnen opgesplitst worden in twee groepen: logische veranderingen en fysieke veranderingen. We hebben voor beide groepen het probleem van operationeel blijven opgelost door eerst een aantal kleine basale veranderingen van de structuur van de zoekboom te definiëren, en daarna de uitvoerbaarheid van deze kleine veranderingen aan te tonen. Deze kleine veranderingen kunnen vervolgens gecombineerd worden tot grote veranderingen.

Het onderzoek in Hoofdstuk 8 “Prototype Implementation” betreft de prestaties van een prototype implementatie van de Globe lokalisatiedienst. We hebben een prototype van de Globe lokalisatiedienst geïmplementeerd om aan te tonen dat een echte implementatie niet de schaalbaarheid van het ontwerp, als gepresenteerd in Hoofdstukken 2–7, aantast. Om de metingen van de prestaties te versimpelen, hebben we alleen gekeken naar de prestaties van individuele (fysieke) knopen.

De metingen laten zien dat een individuele knoop 9 veranderings- of 107 opzoekoperaties per seconde kan uitvoeren. Met deze cijfers en een aantal algemene aannames, kunnen we de prestaties van de Globe lokalisatiedienst als geheel schatten. Deze schattingen laten zien dat in een zoekboom met bijvoorbeeld vier lagen een veranderingsoperatie tussen de 300 en 600 ms duurt en dat een opzoekoperatie tussen de 15 een 650 ms duurt. Deze schattingen ondersteunen onze uitspraak dat een schaalbare lokalisatiedienst gebouwd kan worden. Daarnaast hebben gedetailleerde metingen laten zien dat er ruimte is om de prestaties van de Globe lokalisatiedienst te verbeteren.

De centrale onderzoeksvraag die wij onderzocht hebben in dit proefschrift is: “Hoe kunnen we een wereldwijde lokalisatiedienst bouwen?” We hebben deze vraag beantwoord met het geven van zowel een ontwerp van de Globe lokalisatiedienst als een evaluatie van dat ontwerp. De Globe lokalisatiedienst wordt gekarakteriseerd door de volgende eigenschappen:

- Een gedistribueerde en gepartitioneerde zoekboom die afstanden in het onderliggende netwerk representeert.
- Een communicatiesysteem gebaseerd op remote procedure calls (RPCs) , met een at-least-once fout semantiek en idempotente remote procedures.
- Een caching techniek gebaseerd op het opslaan van verwijzingen naar andere knopen in de zoekboom.
- Een op certificaten gebaseerd mechanisme voor toegangscontrole dat niet-toegestane veranderingen van opgeslagen lokaties voorkomt.
- Een methode om de logische en fysieke structuur van de zoekboom te veranderen terwijl de lokalisatiedienst operationeel blijft.

We baseren onze uitspraak dat het ontwerp van de Globe lokalisatiedienst schaalbaar is op de volgende drie argumenten:

- Het ontwerp is geanalyseerd op het gebied van schaalbaarheid.
- Het gedrag van de Globe lokalisatiedienst is gesimuleerd om de schaalbaarheid en de prestaties van het ontwerp te onderzoeken.
- De prestaties van een prototype van de Globe lokalisatiedienst zijn experimenteel vastgesteld.

Het onderzoek zoals beschreven in dit proefschrift laat ruimte open voor vervolgonderzoek. Op korte termijn is werk aan het prototype nodig omdat het huidige prototype nog niet volledig is. Voor een volledige evaluatie van de Globe lokalisatiedienst is het eerst nodig dat het prototype wordt uitgebreid met alle beschreven functionaliteit. Op de lange termijn is onderzoek nodig naar replicatie- en mobiliteitspatronen van Internet diensten. Kennis van dit soort patronen zal ons in staat stellen het gedrag van de Globe lokalisatiedienst beter te simuleren, en daarmee tot een betere evaluatie van zijn prestaties te komen.

Epilogue

Now that I have finally finished my dissertation, and reached the end of my AiO-schap (i.e., PhD period), it is time for me to look back and acknowledge those people that have supported me along the way in one form or another.

First, I like to thank Maarten van Steen and Andy Tanenbaum for providing me with a pleasant and fruitful research environment. Maarten taught me the "nuts and bolts" of doing research. In the early years, this meant providing structure and guidance on practical matters, and in the later years this meant providing freedom and in-depth discussions. One trait Maarten has not been able to teach me: an ever present, optimistic outlook on the research. I will surely miss that! Andy played the important role of goal keeper: Whenever Maarten and I got too close, too infatuated with our own little inventions and research ideas, Andy would take one critical look and sent us back to the drawingboard, challenging us to improve our ideas.

I would like to thank the members of my reading committee, Frans Kaashoek, Franz Hauck, Frances Brazier, and Evaggelia Pitoura for the effort they put into reviewing this dissertation. I have put their comments to good use. I especially like to thank Frans Kaashoek for playing the role of *referent*. His thorough and in-depth comments have significantly improved the contents of this dissertation.

Ahhh, and what does one do without a supportive office mate? I can only say that Arno was everything I needed in an office mate: quiet and full of answers. Every time I needed to know something about the inner workings of the Vrije Universiteit, I could simply ask Arno without having to lookup anything myself. Such luxury! I'm sorry that his good example of starting the workday early (i.e., before 9:30), never rubbed off on me. I guess getting up on-time will remain a problem for me for some time. Arno, thanks for your support and your patience. And, if you decide to do education again, remember the AiO motto: "Hard zijn en hard blijven!"

During the seven years of being an AiO, a central part of my workday was the lunch break — for me actually more a brunch break — which allowed me to mingle and talk with colleagues I didn't directly work with. The topics of these lunch discussions are too numerous to list, but rest assured, there was always somebody with a bold opinion on something. For fear of offending any member (past or present) of the lunch club by not naming him (or her) explicitly, I will simply thank you all as a group; you'll know whether it applies to you or not.

During my PhD period I was twice given the opportunity to temporarily suspend my research during the summer months, and visit other research institutes. My first visit was in 1997 to the Computer Sciences department of the Friedrich-Alexander-Universität Erlangen-Nürnberg. During this visit, Franz Hauck was my host. My second visit was in 2000 to the IBM Thomas J. Watson Research Center located in Hawthorne, New York. During this visit, Leendert van Doorn was my host. I would like to thank both Franz and Leendert for the possibility to broaden the scope of my research, and look at other interesting research problems.

As the saying goes: "Mens sana in corpore sano." However, computer science research is not really conducive to a healthy body. I therefore had to achieve this goal in an other way: Total Workout (formerly known as *Bommen*). However, motivating yourself twice a week can be difficult. Luckily, there have been people that put some friendly peer pressure on me. Broer, Gerald, Pim, and Mirna, thanks for your support, and I will return to the training floor shortly! And BTW, how many push-ups do I still need to do?

I like to thank Frank Niessink and Michel Oey for agreeing to support me during the final phase of my PhD period: the dissertation defense. Having a *paranimf* with a PhD degree and a *paranimf* with extensive experience (three times!) instills me with confidence. Especially, since Michel has read every chapter of my dissertation at least once. Michel, I hope you are a good *souffleur*.

Finally, I like to thank my family for their support. Since my father has championed the notion of academic excellence from when I was young, I guess, becoming an AiO was more-or-less inevitable. Furthermore, the challenge of having two siblings that already obtained their PhDs [Ballintijn, 1994, 1999], has both inspired me and filled me with dread. A common occurrence of being the youngest child. The upside of being the youngest, however, was that by now my parents knew of the ups and downs of writing a dissertation, and thus knew when to show interest and when to leave their "struggling" AiO-son alone. I like to thank them for both.

Gerco Ballintijn

Amsterdam
September 2003