

Algorithmic Design of the Globe Wide-Area Location Service

Extended version

Maarten van Steen (contact)
Franz J. Hauck

Internal report IR-440
December 1997

Abstract.

We describe the algorithmic design of a worldwide location service for distributed objects. A distributed object can reside at multiple locations at the same time, and offers a set of addresses to allow client processes to contact it. Objects may be highly mobile like, for example, software agents or Web applets. The proposed location service supports regular updates of an object's set of contact addresses, as well as efficient look-up operations. Our design is based on a worldwide distributed search tree in which addresses are stored at different levels, depending on the migration pattern of the object. By exploiting an object's relative stability with respect to a region, combined with the use of pointer caches, look-up operations can be made highly efficient.

Keywords: *distributed systems, naming/location service, mobile computing, object replication, worldwide scalable systems*



vrije Universiteit

Faculty of Mathematics and Computer Science

1 Introduction

As the Internet continues to grow exponentially, the problem of locating people, services, data, software, and machines is becoming more severe. To compound the problem, increasingly many users are no longer tied to a single, fixed access point, but instead are using mobile hardware such as telephones, notebook computers, and personal digital assistants. Applications must therefore take into account that a user will have to be located first in order to deliver any messages [1, 2]. Likewise, the mobile user will possibly also have to find local, nonmobile resources at the location he or she is currently residing (e.g., a local laser printer) [3, 4].

Besides mobility that is tied to mobile hardware, we can also expect software and data to move within a network. For example, to support ubiquitous computing, it will be necessary to move a user's personal environment from one location to another [5]. Another example of software mobility is the active transfer of Web pages to replication servers in the proximity of clients [6, 7]. Likewise, software agents may be roaming the network in search of information, representing their owner at servers, etc. [8]. Finally, with the introduction of Java, mobile code will form an important component of many future Web-based applications [9, 10].

We use the term **mobile object** to collectively refer to any component – implemented in hardware, software, or a combination thereof – that is capable of changing locations. We assume that a mobile object can be distributed or replicated across multiple locations, meaning that there may be several locations where the object resides at the same time. This can be the case, for example, with a whiteboard application shared between a number of mobile users.

The existence of (worldwide) mobile objects introduces a location problem: The need for a scalable facility that maintains a **binding** (i.e., a mapping) between an object's permanent name and its current address(es). Such facilities are normally offered by wide-area naming systems such as the Internet's Domain Name System (DNS) [11], DEC's Global Name Service (GNS) [12], and the X.500 Directory Service [13].

However, existing naming systems are inadequate for mobile objects for two reasons. First, wide-area naming systems assume that name-to-address bindings hardly change. This assumption is necessary to allow effective use of data caches to improve look-up performance. In a mobile environment, however, we must be able to handle the case that bindings change regularly. Second, most naming systems distribute the name space across different globally distributed naming authorities, and subsequently use location-dependent names [14]. Unfortunately, location-dependent names make it harder to handle migration and replication. Each time an object changes location, or whenever a replica is added or removed, we have to adapt the object's name(s) as well. Alternatively, we could change a name into a forwarding pointer, but this has serious scalability problems when applied in worldwide systems.

What is needed is a naming facility that allows bindings to change regularly and which offers complete location transparency to its users. We have recently completed the design of such a facility, which we call a **location service**, as part of the Globe project [15, 16].¹ The Globe location service is designed to handle trillions of mobile objects worldwide. It uses a worldwide distributed search tree in which addresses of an object's present location are stored. All location operations (updating and looking up addresses) are based on the use of globally unique and location-independent object identifiers. The service can be used in combination with traditional naming services, but which should then map user-defined names to object identifiers instead of addresses. Our approach distinguishes itself by (1) scal-

¹Information on the Globe project can be found at <http://www.cs.vu.nl/~steen/globe/>.

ing worldwide and to trillions of objects, (2) allowing objects to frequently update name-to-address bindings, and (3) supporting distributed objects that reside at multiple locations at the same time.

In this paper, we present the basic algorithms for updating and looking up locations. In Section 2 we give an outline of our approach, followed in Section 3 by a detailed description of our algorithms. Several improvements are discussed in Section 4. Related work is presented in Section 5. We conclude and discuss future work in Section 6.

2 Architectural Design

In this section, we outline the architecture of the Globe location service. An overview of our approach can also be found in [17].

2.1 Naming and Locating Objects

A naming and location service maintains a mapping between a user-defined name of an object and that object's location. Traditional naming services generally store name-to-address bindings directly. In other words, each binding consists of a record containing the name and address of an object. In this approach, we are forced to update the binding whenever the object changes its location. Likewise, the binding has to be updated whenever the user decides to change the object's name. Consequently, by storing bindings between a user-defined name and an object's location as records in a database, we create a dependence between two different, and in principle unrelated kinds of updates. For a wide-area system, such a dependence may introduce serious management and scalability problems.

In Globe, we follow a different approach. We separate naming from location issues by introducing a two-layered naming hierarchy. The upper layer deals with hierarchically organized, user-defined, human-readable name spaces. The lower layer deals with keeping track of each object's location independent of how that object is named by its users. The interface between the two layers is formed by **object handles**: a user-defined name is bound to an object handle, which in turn is bound to the address(es) where the object can be found.

An object handle is designed specifically for looking up an object's present location. It contains a *Service-independent Global Unique Identifier* (SGUID) which is similar to a Universal Unique Identifier in DCE [18]. A SGUID is a true object identifier [19]:

1. each SGUID refers to exactly one object
2. each object has exactly one SGUID
3. a SGUID is never reused
4. an object will never get another SGUID than the one initially assigned to it

An object handle will generally obey the same properties, although an object might have several object handles. An object handle may also contain information that can be used to assist in locating the object. An important property of an object handle is its stability: it is assigned once to an object, and remains the same during that object's lifetime, no matter where the object moves to. No two objects ever have the same object handle, even if generated 100 years apart in distant countries.

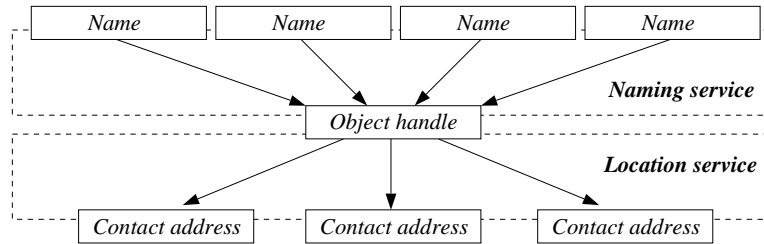


Figure 1: A two-level naming hierarchy that allows an object’s name and contact addresses to be independently changed.

Mapping user-defined names to object handles is done by **anaming service**, and which can be based on existing technology. For example, because object handles do not change, an implementation can make effective use of caching name-to-handle bindings, analogous to the approach followed in DNS [11]. In fact, we can even use TXT records in DNS to implement our name-to-handle bindings.

In contrast, mapping an object handle to a set of addresses is the main task of **alocation service**. In Globe, we adopt a model in which an object offers **contact addresses** to client processes. A contact address describes where and how an object can be reached [15]. A contact address consists of, for example, an IP address, a telephone number, or another kind of address, as well as additional information that identifies the place where the address lies. We allow an object to regularly change its location, that is, to regularly change the binding between its object handle and contact address. In addition, we also provide support for binding several addresses to a single object handle. In this way, it becomes much easier to handle replicated objects. In this model, a mobile, replicated object is characterized by having a set of contact addresses which may change over time.

2.2 General Organization

To efficiently update and look up contact addresses, we organize the underlying wide-area network as a hierarchy of geographical, topological, or administrative **domains**, similar to the organization of DNS. For example, a lowest level domain may represent a campus-wide network of a university, whereas the next higher level domain represents the city where that campus is located. Lowest level domains are also called **leaf domains**. Each domain D is represented by a separate **directory node**, denoted $dir(D)$, leading to a worldwide search tree as shown in Figure 2. Nodes may be internally partitioned for scalability reasons. The internal organization of the location service is entirely transparent to client processes.

A directory node stores information on objects in **contact records**. Each node has a separate contact record per object. A contact record contains a number of **contact fields**, one for each child of the node where the record is stored. A contact address of an object is always stored at exactly one directory node. In addition, a path of forwarding pointers from the root to the node where the address is stored, is established for that object as well. An implication of this design is that we can always locate a contact address of an object by following a chain of forwarding pointers for that object, starting at the root. In practice, we can do much better, as we describe later.

As an illustration, Figure 3 shows part of the search tree storing several contact addresses on behalf of a single object. The domain represented by a node N is denoted $dom(N)$. In Figure 3, node N_0 contains a

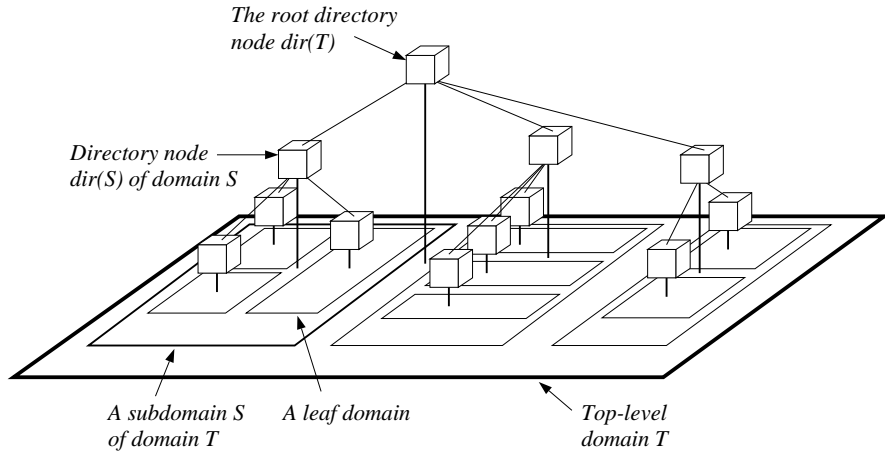


Figure 2: The logical organization of the location service as a worldwide search tree.

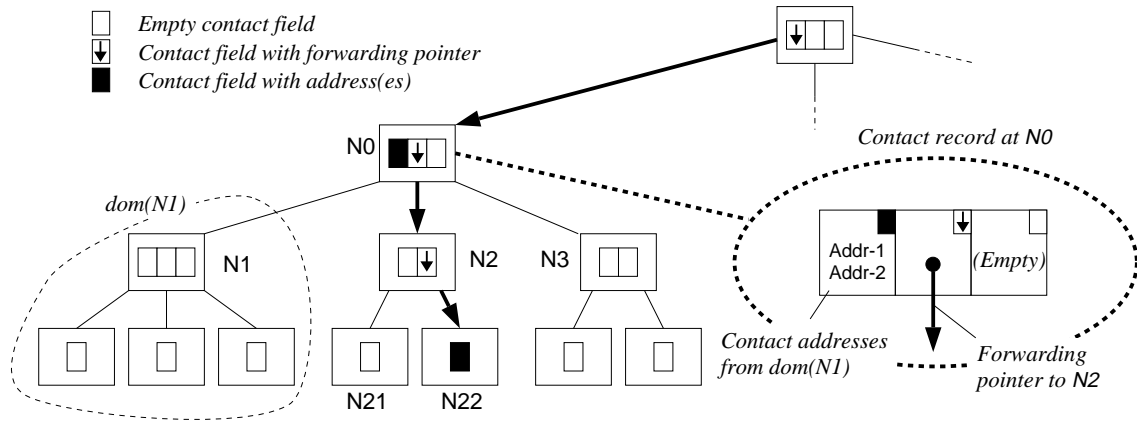


Figure 3: The organization of contact records in the tree for a specific object.

contact record with three contact fields, one for each of its children. The field for child $N1$ contains two contact addresses, which both lie in domain $dom(N1)$. As we motivate in Section 4, although contact addresses are normally stored in leaf nodes, higher level nodes may decide to store addresses as well. We follow the policy that in such cases, higher level nodes have priority over lower level ones. The contact field for child $N2$ contains a forwarding pointer, meaning that somewhere in the subtree rooted at $N2$ there should be at least one other contact address stored for the object. Finally, the contact field for node $N3$ contains no data at all, implying that there are no contact addresses that lie in domain $dom(N3)$. If none of the contact fields of a contact record contains data, the contact record is said to be **empty**.

Storage of addresses and pointers is subject to a number of consistency conditions. In particular, when there are currently no update operations in progress for a specific object O , we require that the following three conditions are met:

- 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 the root to $\text{dir}(D)$.

C2: *For each node N , the contact record for 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 nonempty.*

This means that we do not accept dangling pointers in our tree. In other words, if we follow a forwarding pointer we should eventually find a contact record containing one or more addresses.

C3: *A contact field can contain either a forwarding pointer or contact addresses, but not both.*

Together with the previous conditions, this condition implies that as soon as we encounter a contact field containing contact addresses, we can be sure that we have found all contact addresses that lie in the subdomain represented by that contact field.

When these conditions are met, the tree is said to be **globally consistent** for O . As an example, the tree shown in Figure 3 is globally consistent.

As we discuss below, a contact address that lies in leaf domain D is always inserted or deleted by initiating a request at the directory node $\text{dir}(D)$ of D . To simplify matters, we require that the identity of the leaf domain in which the address lies is encoded in the address. For example, a contact address could be represented by a record containing fields for the type of network address (such as “IPv6”), the actual network address, and a name such as “cs.vu.nl” that identifies the leaf domain where that address lies. In contrast to most network addressing schemes, our contact addresses are thus seen to be location dependent.

2.3 Update Algorithms

We require that an update operation on a globally consistent tree leaves the tree in a global consistent state after its completion (assuming that no other operations for the same object are still in progress). For an insert request initiated at leaf node $\text{dir}(D)$, it is easily seen that global consistency implies that there can be only one node along the path from the leaf node to the root where all addresses from D are stored. In particular, if there is such a node N , then an insert request from *any* leaf domain enclosed by $\text{dom}(N)$ should be forwarded to N .

If there is no node that is already storing addresses from D , we can choose one along the path to the root as long as the global consistency constraints are satisfied. We follow the policy that the highest level node that wants to store addresses from D , without violating global consistency, will be allowed to store addresses. As we explain in Section 4, this policy allows us to construct highly effective caches, even for mobile objects. Note that only those nodes are eligible for storing contact addresses from D which either have an empty contact record, or an empty contact field for a domain that encloses D .

Whenever an insert request arrives at a node that is willing and capable of storing the address, that node will thus have to check whether there is a higher level node along the path to the root where the address should actually be stored. The general approach to inserting an address is illustrated in Figure 4. When an address is to be inserted, the request is propagated to the first directory node where the object is known, which is N_0 in our example. Due to conditions C2 and C3, nodes higher than N_0 cannot store the address and thus need not be considered. Assuming node N_0 does not want to store the address (as we explain below), an acknowledgment is propagated back to the initiating leaf node while at the same time a path of forwarding pointers is established. In our example, both N_1 and leaf node N_2 want to store the address, in which case N_1 will be permitted to do so.

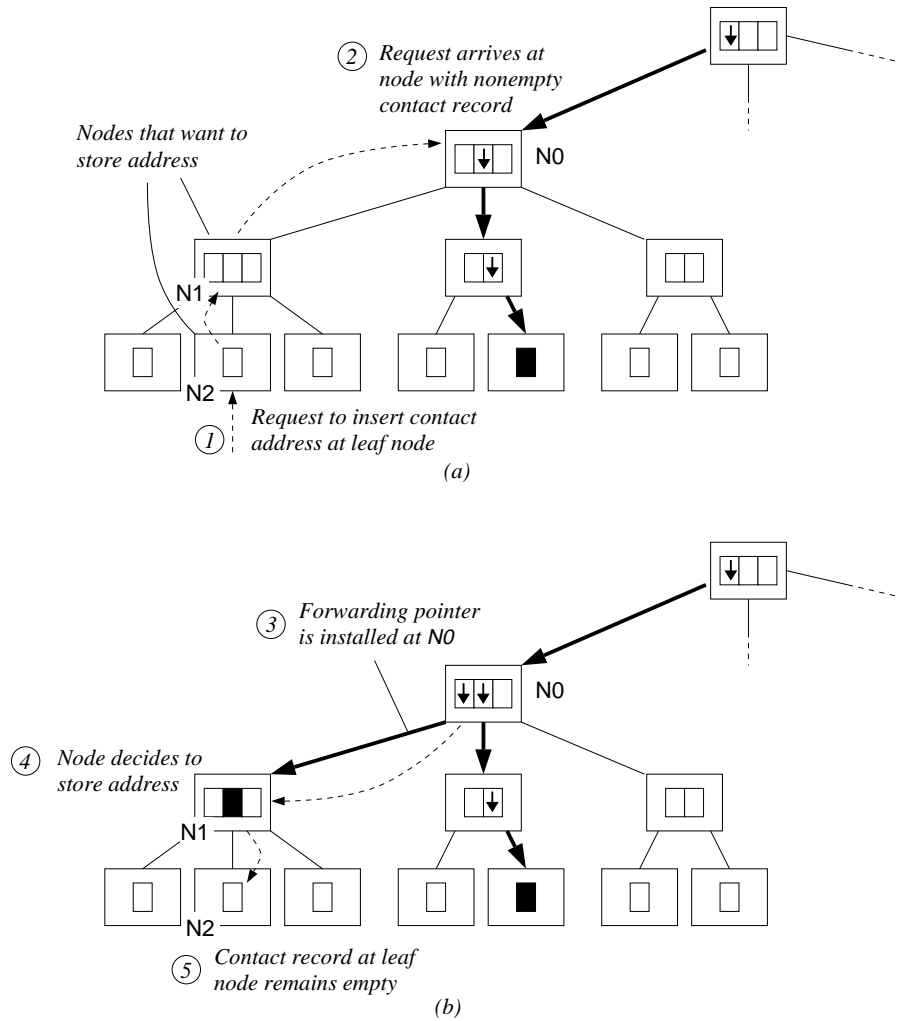


Figure 4: The general approach to inserting a contact address, by which an insertion request propagates upwards to the lowest-level node where the object is known (a), after which a downward path of forwarding pointers is set up (b).

There may be several factors that determine whether or not a node wants to store addresses. For example, as we discuss in Section 4, when an object is highly mobile, meaning that it is inserting and deleting addresses at a relatively high frequency, a node may decide that it is more efficient to store addresses at a higher level node that covers the smallest domain in which the object is moving. This means that, although an insert operation is always initiated at a leaf node, the contact address may actually be stored at a higher level node. There may be other reasons as well that influence the willingness of a node to store addresses. However, we want to decouple our algorithms from such decisions and introduce, for each node, a boolean operation `store_here` that returns `true` if and only if the node wants to store addresses. If, on the path from a leaf node to the root, there is no node willing to store addresses, we follow the policy that addresses are stored in the root node. We allow the outcome of `store_here` to change in the course of time.

Deleting a contact address is straightforward and is done as follows. First, the address is found through

a search path up the tree, starting at the leaf node where the address was initially inserted. Once the contact address has been found, it is removed from its record. If a contact record becomes empty, the parent node is informed that it should delete its forwarding pointer to that record, possibly leading to the (recursive) deletion of forwarding pointers at higher level nodes.

Inserting and deleting contact addresses is targeted toward exploiting locality. Especially when contact addresses already exist in the domain where the operation is being performed, it is seen that the operations can be relatively cheap.

2.4 Look-up Algorithm

Looking up addresses can be done completely independent of the update operations. In this paper, we consider only look-up operations for one contact address; operations that look up several addresses for the same object are easily devised.

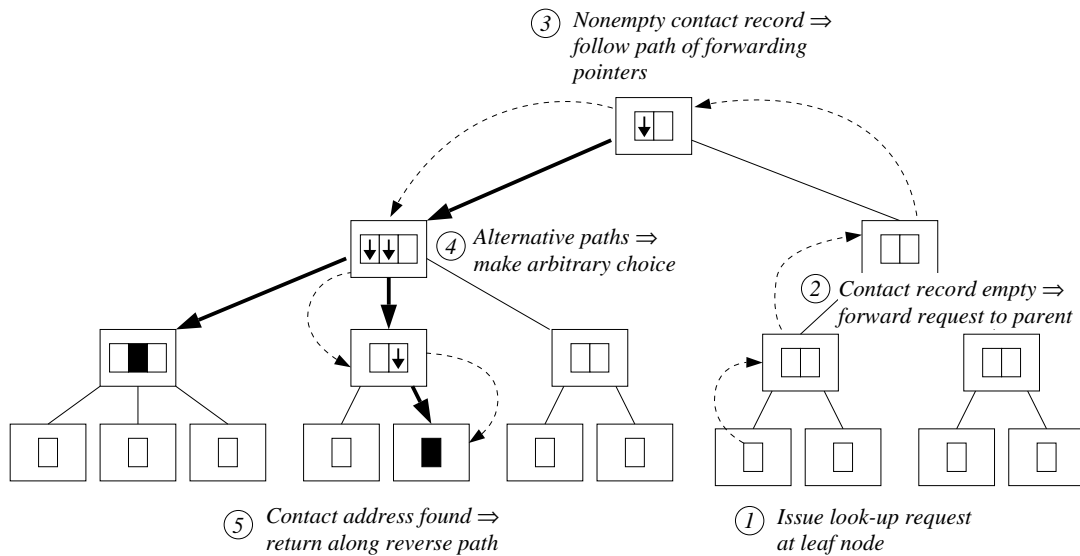


Figure 5: The default approach for looking up a contact address.

We adopt a simple look-up policy. A look-up operation is always initiated at a leaf node (in particular the one in the client's domain), and forwarded along the path to the root until a node is reached having a nonempty contact record. If that record contains a contact address, then the address is returned to the client process. Otherwise, if the record contains only forwarding pointers, a depth-first search is initiated at an arbitrary child, until an address is finally found. This approach is shown in Figure 5.

Again, it is seen that we exploit locality: the look-up operation searches local domains first, and gradually expands to larger domains as long as no contact addresses are found.

3 Algorithmic Design

In this section we concentrate on the algorithmic design of our location service. We first present the basic data structures, after which we discuss in detail the insertion of addresses. Address deletion is then relatively straightforward, as well as our look-up algorithm. In the following, we concentrate only on operations for a single object, as operations for different objects are completely independent.

3.1 Preliminaries

3.1.1 Contact Records

For each directory node, we model an object's contact record as an (indexed) set of contact fields, one field for each child. Each contact field stores either a forwarding pointer, or a set of contact addresses, but never both. A leaf node has exactly one contact field. Adopting an Ada-like notation, we can describe these data types as shown in Figure 6. We assume that each node has a unique identifier of type `NodeID` that can be used as an index for sets of contact fields. An opaque data type `Address` is used to model contact addresses.

```
type ContactField is
  record
    addrSet : set of Address :=  $\emptyset$ ;           -- Set of contact addresses for subdomain
    isPtr : Boolean := false;                   -- True iff contact field is forwarding pointer to child
  end record;
type ContactRecord is set (NodeID) of ContactField; -- Indexed set of contact fields
```

Figure 6: Data structures for storing contact addresses of a single object at a directory node.

3.1.2 Tentatively Available Data

As we make clear in the succeeding sections, update operations gradually propagate through the tree. While doing so, a decision is made *where* to actually store or remove data. For example, our update protocol prescribes that before storing an address `addr` at some node `N`, we first need permission from `N`'s parent. If we wait until that permission is granted, `addr` cannot yet be looked up, despite the fact that we already know that it is a valid contact address. Therefore, it makes sense to make the address *tentatively available* at the node where the operation is currently being performed, without giving guarantees that it will eventually also be stored there. To support tentative availability of updates, we introduce views and view series.

A **view** on a variable `v` is a statement expressing a change to the value of `v`. Evaluating a view leads to the *tentative* execution of the statement, returning the value that `v` would have had if the statement had actually been executed. Evaluating a view on `v` leaves the original value of `v` unaffected; it is like a kind of shadow version. View evaluation takes place only by means of **view series**. A view series associated with a variable `v` is a FIFO-ordered list of views on `v`. The value of a view series is defined as the result of evaluating its views in the order that they have been appended to the series.

This mechanism is best illustrated by an example. In Figure 7, we declare integer variables `x` and `y`, and an integer view series `vx` that is associated with `x`. (The notation $\langle a, b, c \rangle$ denotes a list of elements

(1) <code>x : Integer := 4;</code>	
(2) <code>y : Integer;</code>	
(3) <code>vx : view series of Integer := x;</code>	<code>x = 4; vx = ⟨ ⟩</code>
(4) <code>append view ⟨self := self + 1⟩ to vx; y := vx;</code>	<code>x = 4; y = 5; vx = ⟨x + 1⟩</code>
(5) <code>append view ⟨self := self * 2⟩ to vx; y := vx;</code>	<code>x = 4; y = 10; vx = ⟨x + 1, 2 * x⟩</code>
(6) <code>x := 5; y := vx;</code>	<code>x = 5; y = 12</code>
(7) <code>apply view to vx;</code>	<code>x = 6; y = 12; vx = ⟨2 * x⟩</code>

Figure 7: A simple example of views and view series.

a, b, c , with a being the head of the list.) In line 4, we append a view that expresses an increment of x by 1. The pseudo-variable `self` points to the variable associated with the view series, in this case `x`. We then subsequently assign the value of `vx` to `y`. At that point, the value of `y` is 5, whereas `x` is still 4. In line 5, another view is appended expressing a multiplication by 2, followed by an update of `y`, which now has the value 10. Note that at this point, the value of `vx` is $2 \cdot (x + 1)$. Therefore, if we change the value of `x` to 5, as in line 6, and update `y` again, `y` will become 12.

The view at the head of a view series, that is, the least recently appended one, can be **applied** by evaluating its expression and changing the value of the associated variable accordingly. The view is then removed from the view series. For example, in line 7, we apply the first view to `x`, thereby changing the value of `x` to 6 by incrementing it by 1. At the same time, the view is removed, so that the view series `vx` now reflects only the value $2 \cdot x$. A view can also be directly **removed**, that is, without applying it. Finally, the function `sizeof` returns the length of a given view series.

A contact record for an object O at node N has an associated view series `tentativeCR(O,N)`. Because we consider only operations for a specific pair of object and node, we omit the indices throughout the remainder of our discussion. This view series is an instance of the following data type:

type TentativeRecord is **view series of** ContactRecord ;

As we shall see, all update operations first append a view to a contact record's view series to reflect the intended update. However, this result is still tentative. Later, when the final decision can be made on the update, the previously appended view is either applied, making the result authoritative, or undone by removing the view from the view series. Details are explained in the next section.

3.1.3 Remote Invocations

Our algorithms are based on an RPC mechanism [20], by which a node invokes an operation at its parent, and subsequently blocks until a reply is received. We assume that the execution of an update or look-up operation for a specific object runs to completion or until it blocks, without being pre-empted by competing operations. To ensure correctness of our algorithms, we require that invocation requests and the subsequent responses, are handled in the order that they were issued. How these semantics are implemented is described in [21].

3.2 Address Insertion

The insertion of an address for a specific object is done by two operations:

- `insert_addr` is invoked at a node when that node is requested to store the given address
- `insert_chk` is invoked at a parent node to obtain permission to store the address at the invoking node, or one of its children

It is important to note that whenever either operation is invoked at a specific directory node, it is known at that point that the given address can be used to contact the object. In other words, the address can, in principle, be returned as the result of a look-up operation. The only thing that is not yet known, is exactly at *which* node the address will be stored. For example, when returning to Figure 4, we see that as soon as the insert request is initiated at leaf node N_2 , we can already make the address available to look-up operations from $dom(N_2)$. Likewise, when the request is propagated to N_1 , the address can be made available to look-up requests from N_1 . In both cases, we do not yet know *where* the address will actually be stored. Our insert operations, therefore, can start by making the address tentatively available at the present node without yet having permission from the parent. Making the address tentatively available means that either the address, or a forwarding pointer to the calling node is tentatively stored.

```

(1) procedure insert_addr(caller : NodeID; addr : Address) return (OK, DELETE) is
(2)   viewedCR : TentativeRecord := tentativeCR; -- Make a copy of the current view series
(3)   final_action : (OK, DELETE) := OK;
(4)   -- Start by making the inserted address tentatively available, by appending it to the contact
(5)   -- record's associated view series.
(6)   append view {self(caller).addrSet := self(caller).addrSet + {addr}} to tentativeCR;
(7)   -- Test whether the parent node is to be asked for permission to store the address. This is
(8)   -- necessary when (1) the contact record appeared to be empty or (2) when no authoritative
(9)   -- decision could yet be made.
(10)  if parent ≠ NIL and (empty(viewedCR) or sizeof(viewedCR) > 0) then
(11)    if empty(viewedCR) and not store_here(tentativeCR) then
(12)      -- The contact record appeared to be empty, but the node is not prepared to store the address.
(13)      -- Forward the request to the parent and ensure the appended view is removed.
(14)      parent.insert_addr(thisNode, addr);
(15)      final_action := DELETE;
(16)    else
(17)      -- The node wants to store the address, or may have to because there appear to be other
(18)      -- addresses stored also. Check with the parent whether storing is permitted.
(19)      final_action := parent.insert_chk(thisNode, addr);
(20)    end if
(21)  end if
(22)  if final_action = OK then apply view to tentativeCR;
(23)    else remove view from tentativeCR;
(24)  end if
(25)  return OK;
(26) end insert_addr

```

Figure 8: Insertion of contact addresses.

Operation `insert_addr`

We start with the operation `insert_addr`, which is specified in Figure 8. We assume there is a function `thisNode` that returns the node identifier of the node where the function is called. As mentioned before, the variable `tentativeCR` denotes the view series associated with the object's contact record at the current

node. The operation starts with saving the state of the current contact record in line 2 after which it makes the address available to look-up operations by tentatively adding it to `tentativeCR` in line 6.

As a next step, the node has to check whether and how it should contact its parent. There are three occasions on which the parent needs to be contacted:

- If the contact record was empty when the operation was invoked, the node may choose to store the address. If it is not prepared to store the address, it should pass the request to its parent. This is exactly what happens in lines 11–15. It also means that the previously appended view should be removed again when the call to the parent returns, as expressed in line 15. Note that the address is simply passed to the parent by calling `invoke_addr` again in line 14.
- If the contact record was empty and the node wants to store the address, it will have to ask its parent for permission by invoking `insert_chk` in line 19.
- Likewise, permission is also needed when there are pending requests to the parent, that is, when a number of tentative results from previous operations still exist. In that case, the node cannot take any definitive decision on whether or not to store the address. This situation is also covered by the invocation of `insert_chk` in line 19.

Depending on whether the parent had been called, or what the response was, the operation eventually continues with either turning the previously appended view into authoritative data (line 22), or removing it altogether (line 23).

Operation `insert_chk`

The operation `insert_chk` is invoked at the parent node when the invoking node or one of its (grand)children wants to store the given address. The parent is asked for permission to store the address at one of its (grand)children.

If the parent agrees, it will, in turn, have to obtain permission from the next higher level node, and so on up to the root of the tree. This permission results from our policy that the highest level node that wants to store addresses, may do so, provided global consistency is not violated. Permission is not needed if the parent had already stored a forwarding pointer to the calling child. When the invoked node permits its (grand)child to store the address, it tentatively installs a forwarding pointer to the calling child, thereby making the address available for look-up operations in its domain. The pointer can be only tentatively installed as long as higher level nodes have not yet given their permission for storing the address at some lower level.

Alternatively, the parent may decide that it wants to store the address itself, and that it can do so without violating global consistency. In that case, the invoking child, which will have made the address tentatively available, is instructed to remove the address or its forwarding pointer from its view series. Removal is recursively propagated downwards to the lowest level node where the address has been tentatively stored.

The operation `insert_chk` has a similar structure to `insert_addr` (see Figure 9). It decides whether to tentatively add the given address to its contact record, or tentatively install a forwarding pointer to the calling child (lines 9–14). An address is always added if there are already contact addresses in the corresponding contact field. When the contact field was empty, that is, it also did not contain a forwarding pointer to the calling child, the node may decide to store the address using its `store_here` operation. When

```

(1) procedure insert_chk(caller : NodeID; addr : Address) return (OK, DELETE) is
(2)   viewedCR : TentativeRecord := tentativeCR;
(3)   subRecord : ContactField := viewedCR(caller);
(4)   parent_response, my_response : (OK, DELETE);
(5)   -- If this node already stores addresses, the new address should be stored here as well. This
(6)   -- is also true when the contact record is empty but this node wants to start storing addresses.
(7)   -- In that case, it has priority over the calling child. In all other cases, it will, in principle,
(8)   -- allow its child to store the address and ensures it has a forwarding pointer to the child.
(9)   if subRecord.addrSet  $\neq \emptyset$  or (not subRecord.isPtr and store_here(tentativeCR))
(10)  then append view (self(caller).addrSet := self(caller).addrSet + {addr}) to tentativeCR;
(11)  my_response := DELETE;
(12)  else append view (self(caller).isPtr := true) to tentativeCR;
(13)  my_response := OK;
(14)  end if
(15)  -- Now test whether the parent node is to be asked for permission to store the address. This is
(16)  -- necessary when (1) contact record appeared to be empty or (2) when no authoritative
(17)  -- decision could yet be made.
(18)  if parent  $\neq$  NIL and (empty(viewedCR) or sizeof(viewedCR) > 0)
(19)  then parent_response := parent.insert_chk(thisNode, addr);
(20)  else parent_response := OK;
(21)  end if
(22)  if parent_response = OK then apply view to tentativeCR;
(23)  return my_response;
(24)  else remove view from tentativeCR;
(25)  return DELETE;
(26)  end if
(27) end insert_chk

```

Figure 9: Checking an insert operation with a parent.

an address is (tentatively) added, the calling child must clear its contact record. This is accomplished by replying with DELETE (lines 10–11).

When the invoked node is not going to store the address, it gives the calling child permission to do so instead. The invoked node will not store the address because it either is not prepared to do so, or because it already has a forwarding pointer to the calling child. (Note that whenever a contact field already has a forwarding pointer, it can never decide to store an address. In other words, we discard the outcome of `store_here`.) In any case, it will have to ensure that the address becomes (tentatively) available, by having a forwarding pointer to the caller. The latter is ensured by simply installing the pointer, as is done in lines 12–13.

There are two occasions when the invoked node has to pass the request to its parent:

- When there are still pending requests to the parent that have not been answered yet, the node cannot take an authoritative decision on whether or not to make the address available. In that case, the parent has to be asked for permission as well.
- When the node had an empty contact record when the insert request arrived, this invocation concerns currently the *only* address from the node's domain. In that case, the parent is also unaware of the address, and should be asked for permission, regardless whether the node is prepared to store the address or not.

These two cases are specified in lines 18–21. Finally, depending on the reaction of the parent, the previously appended view is either applied or removed as shown in lines 22–25.

3.3 Address Deletion

Deleting an address is done by a single operation `delete_addr`. The operation must be invoked at the same leaf node where the associated address insertion was initiated. (Note that we assume that the leaf domain in which a contact address lies is encoded in the address. We can thus easily identify the leaf node where the deletion should be initiated.) When a contact record at node `N` becomes empty after deleting an address, the parent node should delete its forwarding pointer `toN`. Removing a pointer at a parent node is handled by `delete_addr` as well, for which case it has an additional boolean parameter `delPtr`. The operation is specified in Figure 10.

```

(1) procedure delete_addr(caller : NodeID; addr : Address; delPtr : Boolean) return (OK,NOTFOUND) is
(2)   viewedCR : TentativeRecord := tentativeCR;
(3)   addrFound : Boolean := (addr ∈ viewedCR(caller).addrSet); -- True iff the address is here
(4)   ptrFound : Boolean := (delPtr and viewedCR(caller).isPtr); -- True iff there is a pointer to the caller
(5)   -- If either the address is (tentatively) stored at this node, or a (tentative) pointer to the calling
(6)   -- node exists, the operation will have to delete the address or pointer, respectively. Again, the
(7)   -- results of the delete operation can be made available immediately.
(8)   if addrFound or ptrFound then
(9)     if addrFound
(10)      then append view {self(caller).addrSet := self(caller).addrSet - {addr}} to tentativeCR;
(11)      else append view {self(caller).isPtr := false} to tentativeCR;
(12)    end if
(13)    -- When the contact record is now empty we know that the parent has a pointer installed to this node.
(14)    -- In that case, request the parent to delete it.
(15)    if parent ≠ NIL and empty(tentativeCR) then
(16)      parent.delete_addr(thisNode, addr, true);
(17)    end if
(18)    -- Unconditionally apply the previously appended view, i.e., remove either the address or the
(19)    -- forwarding pointer.
(20)    apply view to tentativeCR;
(21)    return OK;
(22)  elsif parent ≠ NIL and (empty(tentativeCR) or sizeof(tentativeCR) > 0) then
(23)    return parent.delete_addr(addr, false);
(24)  else return NOTFOUND;
(25)  end if
(26) end delete_addr

```

Figure 10: Deletion of contact addresses.

Completely analogous to making newly inserted addresses tentatively available, we can also immediately announce that an address or forwarding pointer will be removed. In other words, as soon as a node `N` is requested to delete an address or forwarding pointer, it can do so without waiting for its parent to have completed the operation. Deletion takes place by appending a view by which the address or forwarding pointer is removed from the contact record. In this way, we even achieve that a previously inserted address for which the insert operation has not yet fully completed, that is, the address is yet only tentatively available at a node, is immediately made unavailable again to look-up operations at that node. Such effects are important in wide-area systems. An alternative, by which a deletion can

come into effect only after the associated insertion has completed, is generally unacceptable due to unpredictable delays for the completion of an operation.

Therefore, the operation `delete_addr` starts with undoing the effects of the previous insert operation (lines 3–12). First, it checks whether it stores the address (line 3) or forwarding pointer (line 4), after which a view is appended reflecting the respective removal (lines 10–11).

There are two occasions in which the parent should be called as well:

- If the contact record was already empty, or when it became empty on account of the current delete, the parent node should remove its forwarding pointer to the current node. This situation is specified in lines 15–17 for the case that record became empty, and in line 23 for the case that it already was empty.
- If there were pending operations to the parent, the node does not yet know what the final situation will be when all previous requests have been processed. Therefore, the parent must be informed about the deletion as well. This situation is also expressed in line 23.

3.4 Address Look-ups

An important design issue for our location service is that we wish to make update results available as soon as possible. This is important in a wide-area system, where propagations of updates may take a relatively long time due to network and node failures. Therefore, look-ups operate on tentatively available data, that is, the value of view series, rather than on the authoritative data of contact records.

This policy works fine in a tree that is globally consistent, and even in a tree where some addresses have been made tentatively available only. Problems arise when some addresses are being deleted concurrently with look-up operations, for in that case, we may decide to follow a path of forwarding pointers that is in the process of being deleted. In that case, we also adopt a simple solution. If a path has been followed without success, we simply continue the look-up operation in another path, if possible. If all such attempts fail, the look-up operation proceeds with the next higher level node on the path to the root.

Our operation `lookup` is given in Figure 11. It starts with checking whether the current node has a nonempty contact record (line 4). If so, it tries to select an arbitrary contact field containing addresses. This is expressed by the **choose any** statement in line 7, which, in this case, takes an index as a free variable and tries to match that in the expression following the **with** keyword.

If the selection succeeded, the operation subsequently selects an arbitrary address from that contact field (again expressed as a **choose any** statement), and returns the address as the result to the calling node (lines 8–10).

On the other hand, if there were no addresses in the contact record, the look-up operation continues by following an arbitrary path of forwarding pointers in one of the subtrees rooted at a child. Because each of these paths may be in the process of being deleted, all contact fields containing a forwarding pointer are checked (line 12). As soon as an address has been found in one of the subtrees, the operation stops by returning that address (line 14).

If no address could be found, we need to continue the look-up operation at a higher level node (line 19). This makes sense only when the operation was initially called by one of the children, or by a client process, that is, `caller ≠ parent`. Otherwise, when no address was found, we have reached the root of the

```

(1) procedure lookup(caller : NodeID) return Address is
(2)   addr : Address := NIL;
(3)   -- First check whether this node has any information on the object.
(4)   if not empty(tentativeCR) then
(5)     -- In principle, we should be able to find something here. Check whether any address is
(6)     -- (tentatively) stored in this contact record. Otherwise, follow paths in the subtrees.
(7)     choose any child with tentativeCR(child).addrSet  $\neq \emptyset$ ;
(8)     if child  $\neq$  NIL then -- An address has been found. Any stored address will do.
(9)       choose any addr with addr  $\in$  tentativeCR(child).addrSet;
(10)      return addr;
(11)   else -- Check any downward path. If the path is being deleted, select a next one.
(12)     foreach child with (child  $\neq$  caller) and (tentativeCR(child).isPtr = true) loop
(13)       addr := child.lookup(thisNode);
(14)       if addr  $\neq$  NIL then return addr end if
(15)     end loop;
(16)   end if
(17) end if
(18) if addr = NIL and caller  $\neq$  parent
(19)   then return parent.lookup(thisNode)
(20)   else return addr
(21) end if
(22) end lookup

```

Figure 11: Looking up a single contact address.

tree, and NIL, which is the present value of `addr` can be returned (line 20). If we did find an address, we simply return that value.

3.5 Discussion

If we ignore the use of view series, our algorithms are relatively straightforward and strongly resemble standard (recursive) implementations for search tree algorithms. The intricacies mainly come from the fact that we wish to make results available as soon as possible. This explains why every operation starts with appending its anticipated result to the view series associated with the current contact record. Effectively, view series allow us to propagate update results in increasingly expanding domains *before* the update has been fully completed. For a wide-area system, the availability of such tentative data is essential, as it may take considerable time before results become authoritative.

To illustrate the benefit of our approach, assume the root node is temporarily unreachable due to a network or node failure. In that case, our location service is temporarily partitioned into a number of subtrees (one for each child of the root node). However, each subtree continues to operate normally, although operations requested to be invoked at the root node will experience a significant delay. By additionally maintaining the order of invocations through view series, we, at worst, experience performance failures. Clearly, the look-up operation needs to be improved, as it is unacceptable that a client must wait until the tree recovers from a failure. Long or indefinite waiting can easily be dealt with by using time-out mechanisms.

To assess the correctness of our algorithms, we initially expressed our update and look-up operations in the protocol verification language Promela [22], and conducted a number of state space searches. After an initial design phase, we constructed formal proofs of correctness. The latter can be found in

the appendix.

4 Improvements

There are several ways in which we can improve the working of the location service described so far. One important optimization consists of adding caches. Another is taking object characteristics into account concerning locality.

4.1 Placement of Contact Addresses

By default, a contact address is stored at the leaf node where it is inserted. However, this may not always be the best choice. Consider the situation that an object is regularly moving between two leaf domains $L1$ and $L2$. Let D denote the lowest level domain that covers both leaf domains. Each time the object moves from $L1$ to $L2$, the location service creates and deletes a path of forwarding pointers from the directory node $\text{dir}(D)$ of D to the leaf nodes $\text{dir}(L1)$ and $\text{dir}(L2)$, respectively. When the object is moving regularly, it makes sense to store the contact address in the object's contact record at $\text{dir}(D)$. For example, by maintaining only the path from the root to $\text{dir}(D)$, we can save on costs for path maintenance.

In addition, there is another advantage of storing addresses at $\text{dir}(D)$. We know that, although the set of addresses stored at $\text{dir}(D)$ may change, the *place* where these addresses are stored is now stable. This permits us to effectively shorten search paths by caching pointers to contact records. Specifically, we cache a pointer to the directory node containing a contact address, at each node of the search path when returning the answer to the leaf node where a look-up request originated, as shown in Figure 12.

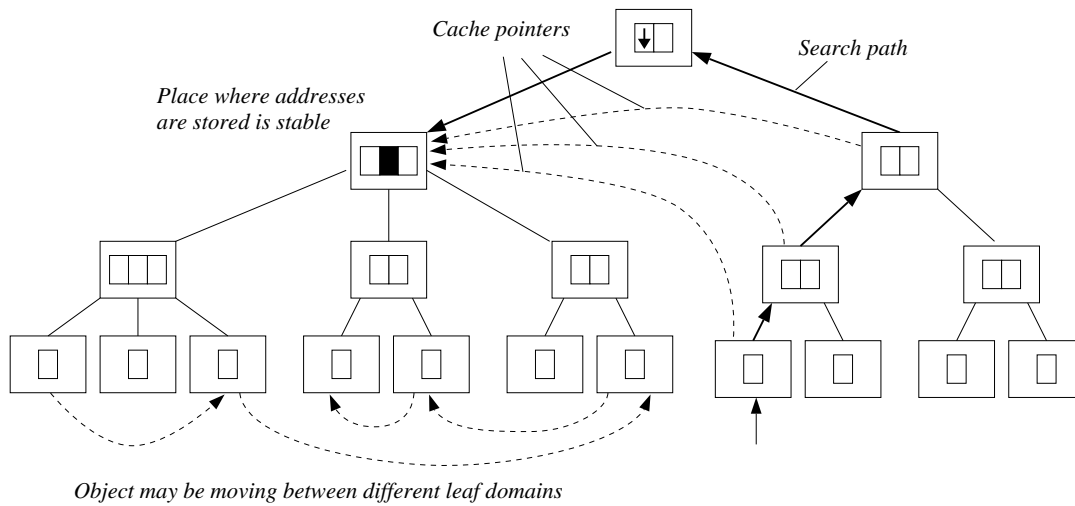


Figure 12: Caching pointers to a stable location, even as the object moves.

We now have the situation that the object which is moving between leaf domains can be easily located by looking up its present address in the node $\text{dir}(D)$ representing the smallest domain in which all its

movements take place. By caching a pointer to $\text{dir}(D)$, the object may be tracked by just two successive look-up operations (assuming a cache hit at the leaf node): the first one at the leaf node servicing the requesting process, and the second one at $\text{dir}(D)$. This is a considerable improvement over existing approaches.

4.2 Object Characteristics

So far, we have been implicitly assuming that objects will want to move between arbitrary leaf domains, and likewise, that look-up requests may come from arbitrary leaf domain as well. Of course, this will generally not be the case. For example, we may know that a representative from some local firm may generally be reached in the area where that firm is located. Moreover, we may even expect that most look-up requests come from that area as well.

In cases such as these, we should take advantage of knowing an object's behavior by making the associated object handle location dependent. If we know an object is generally restricted to some domain D , we encode the identity of $\text{dir}(D)$ into the object handle. Whenever the object updates its set of contact addresses, it proceeds as usual, although addresses will never be stored at a higher level node than $\text{dir}(D)$. Likewise, when the object's addresses are to be looked up by a process outside $\text{dom}(D)$, we start our search at $\text{dir}(D)$ instead of at the leaf node local to the requesting process.

Whenever the object moves outside domain D , we simply store its contact address in $\text{dir}(D)$. What is seen here, is that by encoding a domain into an object handle we are essentially combining our general update and look-up policies with a home-based approach as often proposed for mobile hosts. In those cases that $\text{dir}(D)$ is a leaf node, the combined approach reduces entirely to home-based policies.

5 Related Work

We have made a strict separation between a naming service which is used to organize objects in a way that is meaningful to their users, and a location service which is strictly used to contact an object given a unique identifier. Naming services can be used for finding information based on the *meaning* of a name, as is often used for Internet resource discovery services [23]. In our scheme, information retrieval would start with finding relevant names, retrieving the associated object handles, and having the location service return contact address for each object that was found to be potentially interesting.

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

Mobile Computing

So far, much research has concentrated on *mobile computing* which is generally based on a model in which users migrate between different network locations. Usually, mobility in these cases is tied to mobile hardware such as hand-held telephones, personal digital assistants, and notebook computers. An implicit assumption underlying mobile computing is that the mobile object is always at precisely

one location. Replication is less an issue, except when dealing with fault tolerant issues as, for example, in the case of disconnected file operations [24].

Location management in mobile computing generally follows a home-based approach. This means that the system assumes that there is always *ahome location* that keeps track of the object's current location. Once the present location has been found through the home location, messages can be redirected. This is, for example, the way that mobile IP works [25]. PCNs often work with a two-level search tree in which the second level consists of Visitor Location Registers that contain addresses of visiting hosts in the current region. A distinctive feature of our approach compared to PCNs, is that we have several levels allowing us to exploit locality more effectively by inspecting succeeding expanded regions at linearly incrementing costs.

The main drawback of a home-based approach is that it does not scale well to worldwide systems. First, having to contact a possible distant home location while the object may actually be very near to the calling process is not efficient: all locality aspects are neglected. Second, the approach cannot adequately handle long living objects, as the home location must remain responsible for all its objects forever. This is also true for the situations in which an object has permanently moved to another location, even perhaps decades ago. As a consequence, assigning a lifetime telephone number is hard to realize efficiently with home-based approaches.

As an alternative, there are several proposals based on a hierarchically organized distributed database. A straightforward solution without any caching facilities and in which addresses are always stored in leaf nodes is described in [26]. Awerbuch and Peleg [27] propose a solution in which a moving object leaves a forwarding pointer which is removed only after a considerable distance has been traveled. In this way, a trade-off between costly update operations and scalable look-ups is achieved.

Jain [28] uses an approach to caching that is somewhat similar to ours. He also builds a hierarchical database in which the leaf nodes contain contact addresses, and intermediate nodes pointers similar to ours. Once an object has been located, a pointer to a node covering the domain in which the object is moving can be cached at nodes on the reversed search path. Our approach is different in that the address of frequently moving objects is stored at a higher-level node instead of just a pointer. Consequently, our look-up and update operations appear to be cheaper.

Alternatively, update and look-up strategies can be dynamically adapted to a user's migration pattern as proposed by Krishna et al. [29]. In contrast, we propose to adapt the tree on a per-object basis by allowing addresses to be stored at higher levels when necessary. Our update and location policies remain the same. To avoid global look-ups that may involve many hops, Jannink et al. [30] propose to selectively replicate user profiles. This comes very close to allowing an object to have several contact addresses stored by the location service. In our approach, however, we let the object decide whether or not it wants to provide several contact addresses.

Using a hierarchically distributed database leads to the question when and how updates are propagated through the tree. In most cases, an update becomes visible when it has been completed. For wide-area systems, this approach is not acceptable because update propagation is slow. Instead, the results of update operations should be made available as soon as possible. Similar, in wide-area systems, we cannot accept that an operation is delayed until a previous one is completed. To solve these problems, we introduced view series that are used to implement a notion of tentative data. Our mechanism resembles queued RPCs as used in the Rover toolkit [31], except that we maintain the ordering of invocations. In this sense, view series are comparable to sender-based message logging used for recovery from node and network failures as explained in [32].

Mobile Object Systems

An implicit assumption that location management services for mobile computing are often making, is that the object moves *gradually* through the network. For this reason, many algorithms are seen to work well because updates need not be propagated through the entire distributed database. In contrast to systems for mobile computing, mobile-object systems often deal with *mobile computations*. In these cases, one can imagine users to be fairly immobile, and that instead objects move between locations for reasons of load balancing, dynamic replication, etc. An important difference with mobile computing, is that objects travel at a speed dictated by the network, and may pop-up virtually anywhere. This requires a highly flexible approach to locating objects.

Mobile objects have mainly been considered in the context of local distributed systems. In Emerald, mobile objects are tracked through chains of forwarding pointers, combined with techniques for shortening long chains, and a broadcast facility when all else fails [33]. Such an approach does not scale to worldwide networks. An alternative approach to handle worldwide distributed systems is the Location Independent Invocation (LII) [34]. By combining chains of forwarding references, stable storages, and a global naming service, an efficient mechanism is derived for tracking objects. Most of the applied techniques are orthogonal to our approach, and can easily be added to improve efficiency. However, the global naming service, which is essential to LII, assumes that the update-to-lookup ratio is small. We do not make such an assumption.

A seemingly promising approach that has been advocated for large-scale systems are SSP chains [35]. The principle has been applied to a system called Shadows [36]. SSP chains allow object references to be transparently handed over between processes. In essence, a chain of forwarding pointers is constructed from an object reference to the object. Consequently, there is no need for any location service because an object reference can always be resolved through the chain of pointers. A serious drawback of this approach is that it neglects locality, making it hard to apply to worldwide systems.

6 Conclusions and Future Work

The Globe location service provides a novel approach to locating objects in mobile computing and computation. Although the service has yet to be extensively tested in practice, simulation experiments and local implementations indicate that the service can scale efficiently worldwide. An important component of the service is formed by pointer caches. Further research and experimentation is needed to see whether and how our caching policy can indeed be effectively and efficiently deployed.

We are currently developing a prototype implementation of directory nodes that can be easily tested on the Internet. To come to that point, our research is currently concentrating on minimal support for fault tolerance and security. We initially concentrate on an implementation that can support mobile and replicated Web pages, and which can be seamlessly integrated with existing Web browsers.

References

- [1] G. Forman and J. Zahorjan. "The Challenges of Mobile Computing." *Computer*, 27(4):38–47, Apr. 1994.

- [2] T. Imielinski and B. Badrinath. "Mobile Wireless Computing: Challenges in Data Management." *Commun. ACM*, 37(10):19–28, Oct. 1994.
- [3] B. Jacob and T. Mudge. "Support for Nomadism in a Global Environment." In *Proc. Workshop on Object Replication and Mobile Computing*, San Jose, CA, Oct. 1996. ACM.
- [4] B. C. Neuman, S. S. Augart, and S. Upsani. "Using Prospero to Support Integrated Location-Independent Computing." In *Proc. First Symp. on Mobile and Location-Independent Computing* pp. 29–34, Cambridge, MA, Aug. 1993. USENIX.
- [5] M. Weiser. "Some Computer Science Issues in Ubiquitous Computing." *Commun. ACM*, 36(7):74–83, July 1993.
- [6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. "Enhancing the Web's Infrastructure: From Caching to Replication." *IEEE Internet Comput.*, 1(2):18–27, Mar. 1997.
- [7] J. Gwertzman and M. Seltzer. "The Case for Geographical Push-Caching." In *Proc. Fifth HOTOS*, Orcas Island, WA, May 1996. IEEE.
- [8] C. G. Harrison, D. M. Chess, and A. Kershenbaum. "Mobile Agents: Are They a Good Idea." Technical Report, IBM T.J. Watson Research Center, Yorktown Heights, NY, Mar. 1995.
- [9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA., 1997.
- [10] A. Wollrath, J. Waldo, and R. Riggs. "Java-Centric Distributed Computing." *IEEE Micro*, 17(2):44–53, May 1997.
- [11] P. Mockapetris. "Domain Names - Concepts and Facilities." RFC 1034, Nov. 1987.
- [12] B. Lampson. "Designing a Global Name Service." In *Proc. Fourth ACM Symposium on Principles Of Distributed Computing*. ACM, 1985.
- [13] S. Radicati. *X.500 Directory Services: Technology and Deployment*. International Thomson Computer Press, London, 1994.
- [14] D. Cheriton and T. Mann. "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Trans. Comp. Syst.*, 7(2):147–183, May 1989.
- [15] P. Homburg, M. van Steen, and A. Tanenbaum. "An Architecture for A Scalable Wide Area Distributed System." In *Proc. Seventh SIGOPS European Workshop*, pp. 75–82, Connemara, Ireland, Sept. 1996. ACM.
- [16] M. van Steen, P. Homburg, and A. Tanenbaum. "The Architectural Design of Globe: A Wide-Area Distributed System." Technical Report IR-422, Vrije Universiteit, Department of Mathematics and Computer Science, Mar. 1997.
- [17] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, pp. 2–7, Jan. 1998.
- [18] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Sebastopol, CA., 1992.
- [19] R. Wieringa and W. de Jonge. "Object Identifiers, Keys, and Surrogates - Object Identifiers Revisited." *Theory and Practice of Object Systems*, 1(2):101–114, 1995.
- [20] A. Birrell and B. Nelson. "Implementing Remote Procedure Calls." *ACM Trans. Comp. Syst.*, 2(1):39–59, Feb. 1984.
- [21] G. Ballintijn, M. Sandberg, and M. van Steen. "Scheduling Concurrent RPCs in the Globe Location Service." In *Proc. Third ASCI Annual Conf.*, pp. 28–33, Heijen, The Netherlands, June 1997.
- [22] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, N.J., 1991.
- [23] K. Obraczka, P. Danzig, and S.-H. Li. "Internet Resource Discovery Services." *Computer*, 26(9):8–22, Sept. 1993.

- [24] J. Kistler. *Disconnected Operations in a Distributed File System*, volume 1002 of *Lect. Notes Comput. Sc.* Springer-Verlag, Berlin, 1996.
- [25] C. Perkins. “IP Mobility Support.” RFC 2002, Oct. 1996.
- [26] J. Wang. “A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems.” *IEEE J. Selected Areas Commun.*, 11(6):850–860, Aug. 1993.
- [27] B. Awerbuch and D. Peleg. “Online Tracking of Mobile Users.” *J. ACM*, 42(5):1021–1058, Sept. 1995.
- [28] R. Jain. “Reducing Traffic Impacts of PCS using Hierarchical User Location Databases.” In *Proc. Int’l Conf. on Comm.* IEEE, 1996.
- [29] P. Krishna, N. Vaidya, and D. Pradhan. “Location Management in Distributed Mobile Environments.” In *Proc. Third Int’l Conf. on Parallel and Distributed Information Systems* pp. 81–88, Austin, TX, Sept. 1994. IEEE.
- [30] J. Jannink, D. Lam, N. Shivakumar, J. Widom, and D. Cox. “Efficient and Flexible Location Management Techniques for Wireless Communication Systems.” In *Proc. Second Int’l Conf. on Mobile Computing and Networking*, White Plains, NY, Nov. 1996. ACM/IEEE.
- [31] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. “Mobile Computing with the Rover Toolkit.” *IEEE Trans. Comput.*, 46(3):337–352, Mar. 1997.
- [32] D. Johnson and W. Zwaenepoel. “Sender-Based Message Logging.” In *Proc. 17th Annual International Symposium on Fault-Tolerant Computing* pp. 14–19, Pittsburgh, PA, July 1987. IEEE.
- [33] E. Jul, H. Levy, N. Hutchinson, and A. Black. “Fine-Grained Mobility in the Emerald System.” *ACM Trans. Comp. Syst.*, 6(1):109–133, Feb. 1988.
- [34] A. Black and Y. Artsy. “Implementing Location Independent Invocation.” *IEEE Trans. Par. Distr. Syst.*, 1(1):107–119, Jan. 1990.
- [35] M. Shapiro, P. Dickman, and D. Plainfossé. “SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection.” Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.
- [36] S. Caughey and S. Shrivastava. “Architectural Support for Mobile Objects in Large-Scale Distributed Systems.” In L.-F. Cabrera and M. Theimer, (eds.), *Proc. Fourth Int’l Workshop on Object Orientation in Operating Systems*, pp. 38–47, Lund, Sweden, Aug. 1995. IEEE.

A Correctness

To prove the correctness of our algorithms, we first consider the situation that an update request is accepted only when the previous one has been completed. Proving the correctness in this case is relatively straightforward. We then show that in the concurrent situation, the scheduling strategy at a node does not affect the correctness of our algorithms for the sequential case. In particular, we show that it makes no difference whether a node first schedules and executes requests from its children, or that it first handles all outstanding replies from its parent. The latter scheduling policy corresponds to implementing sequential, nonconcurrent invocations. Effectively, we claim that concurrent operations serialized by means of view series, do not affect the correctness of our algorithms.

A.1 Sequential Invocations

We first consider strict sequential invocations and show that invoking the insert and delete operations on a globally consistent tree will bring the tree back into a globally consistent state on completion.

In the following, let $\text{parent}(N)$ denote the parent node of node N . Assume that the tree is initially in a globally consistent state.

Lemma 1 *If N is a directory node that contains a contact address from region R , then all insert operations invoked at a leaf node in region R will store their contact address at N .*

Proof: From Figure 9 it follows that invoking `insert_chk` at N will always return `DELETE`. Consequently, no child of N will be allowed to store an address from R . From Figure 8 it follows that `insert_addr` is invoked only at N if no descendant wants to store the address. Because the tree is globally consistent, $\text{parent}(N)[N].\text{ptr} = \text{true}$, so that invoking `insert_chk` at $\text{parent}(N)$ will always return `OK`. This means that no ancestor of N will store the address on account of invoking `insert_chk`. Both insert operations `insert_addr` and `insert_chk` will always add the address and immediately return without asking for the parent's confirmation. ■

Lemma 2 *When an insert operation has completed, there is exactly one node N in the invocation chain where the new address was stored.*

Proof: We first show that there is at most one node N where the new address is stored. First, from Figure 9, we see that a node that decides to store an address will always forbid the calling node to store that address as well, by returning `DELETE`. Furthermore, observe that `insert_addr` is called only when the calling node cannot or does not want to store the address. In the case of either `insert_addr` or `insert_chk`, adding the address to the present set of contact addresses is done only when (1) there was no call to a parent, or (2) the parent responded with an `OK`. We conclude that there is at most one node where the address is actually stored.

To show that there is at least one node that will store the address, we may assume, on account of the previous lemma, that there is presently no node in the invocation that stores a contact address. Let M be the first node that is willing to store the address. This means that `insert_addr` has been called at M , and M will now call `insert_chk` at its parent. If no higher level node wants to store the address, all ancestors of M are seen to install a pointer, and the parent of M eventually returns an `OK`. In that case,

the address is stored at M . A similar reasoning shows that if one of M 's ancestors wants to store the node, the highest level one will actually succeed. Finally, if no node is prepared to store the address, `insert_addr` will eventually be called at the root, who has no other choice than to store the address. ■

This brings us to the following conclusion:

Corollary 1 *Let $\langle o_1, \dots, o_n \rangle$ be a series of different insert and delete operations all initiated at the same leaf node, such that (1) a delete operation o_j always applies to an address that was previously inserted by an insert operation o_i , with $i < j$, and (2) for any sub-series $\langle o_1, \dots, o_k \rangle$ with $k \leq n$, the number of insert operations is larger than the number of delete operations. Then, there is exactly one node N where all addresses inserted by $\langle o_1, \dots, o_n \rangle$ are stored.*

We can now prove the following theorem:

Theorem 1 *After the completion of an insert request for an address $addr$, the tree is left in a globally consistent state.*

Proof: We need to consider only the chain of invocations from the leaf node where the insertion is initiated towards the root of the tree. Let N be the node where the address is eventually stored, and let T denote the highest level node in the invocation chain. For two nodes P and Q on the invocation chain, we write $P > Q$ if P is at a higher level than Q .

First, assume there is a node $M < N$ with a nonempty contact record. This record must have been empty before the insertion request for $addr$, because invoking either `insert_addr` or `insert_chk` at M would never result in a call to `parent(M)` for the condition `empty(viewedCR) or sizeof(viewedCR) > 0` evaluates to `false` during either invocation. In other words, we would have $M = T$. Therefore, M can have a nonempty contact record only on account of the insertion of $addr$. However, it is not hard to see that storing $addr$ at N , leads to a chain of `DELETE` responses, resulting in the removal of any previously appended view. This also means that the appended view at M is removed, making its contact record empty again. We conclude that all nodes at a lower level than N are unaffected by the insertion request.

Let $N \leq M \leq T$. An analogous reasoning shows that if N already had a nonempty contact record that $N = M = T$. This means that no nodes besides N are affected by the insertion. In particular, we have that $addr$ is simply added to the contact record at N , thus leaving the tree in a globally consistent state. If N had an empty contact record, `parent(N)` is requested to invoke `insert_chk`. The address can be stored at N only if `parent(N)` returns an `OK`, which can happen only if `parent(N)` appended a view containing a pointer to N . Moreover, `parent(N)` will need agreement from its parent, which is again obtained by requesting invocation of `insert_chk`. The end result is that all nodes $N < M \leq T$ recursively install a forwarding pointer to their respective child on the invocation chain, and that N adds $addr$ to its contact record, leaving the tree in a globally consistent state. ■

Lemma 3 *Let N be the node containing a contact address $addr$. The delete operation for $addr$ will remove the address from N .*

Proof: Because the delete operation is initially invoked at the same leaf node where `insert_addr` for $addr$ was called, N lies on the same path from the leaf node to the root as the invocation chain for the delete operation. Each node $M < N$ on the invocation chain will have an empty contact record, or otherwise the

tree would never have been globally consistent. Consequently, the invocation chain will at least reach node N . There, `delete_addr` appends a view that removes the address, and which is later unconditionally applied. ■

Lemma 4 *When a contact record at N becomes empty, $\text{parent}(N)$ will remove its pointer to N .*

Proof: From Figure 10 it can be seen that as soon as the contact record at N appears to be empty as the result of appending a view by which the last address is removed, $\text{parent}(N)$ is requested to invoke `delete_addr` with parameter `ptr` set to `true`. Because the tree was globally consistent, the parent will find `ptrFound` to be `true`, so that it will tentatively remove its pointer to N . Again, that view is unconditionally applied, possibly after having called its own parent. ■

These two lemmas bring us to the following conclusion:

Theorem 2 *After the completion of a delete operation, the tree is left in a globally consistent state.*

Proof: We first show that if N is a node containing a contact address `addr`, then `addr` will be removed from N . Because the delete operation is initially invoked at the same leaf node where `insert_addr` for `addr` was called, N lies on the same path from the leaf node to the root as the invocation chain for the delete operation. Each node $M < N$ on the invocation chain will have an empty contact record, or otherwise the tree would never have been globally consistent. Consequently, the invocation chain will at least reach node N . There, `delete_addr` appends a view that removes the address, and which is later unconditionally applied.

Next, we show that when a contact record at N becomes empty, $\text{parent}(N)$ will remove its pointer to N . From Figure 10 it can be seen that as soon as the contact record at N appears to be empty as the result of appending a view by which the last address is removed, $\text{parent}(N)$ is requested to invoke `delete_addr` with parameter `ptr` set to `true`. Because the tree was globally consistent, the parent will find `ptrFound` to be `true`, so that it will tentatively remove its pointer to N . Again, that view is unconditionally applied, possibly after having called its own parent. ■

A.2 Concurrent Invocations

We now prove the correctness of our algorithms in the face of serialized concurrent operations as explained in Section 3.

The state of a node is defined as the possibly tentative value of its contact record. Let $\langle o_1, o_2, \dots, o_n \rangle$ be a series of invocation requests at node N for which the associated operation need yet to be invoked. The preprocessing section of an operation consists of the actions until calling an operation at the parent node. The postprocessing section is defined analogously. Denote by $\sigma(S_0, \langle o_1, \dots, o_n \rangle)$ the authoritative state of node N after completing all operations o_i ($1 \leq i \leq n$) starting in the initial and authoritative state S_0 . Likewise, let $\sigma_{pre}(S_0, \langle o_1, \dots, o_n \rangle)$ denote the tentative state of node N after having only pre-processed operations o_i based on the initial state S_0 . We omit the brackets “ \langle ” and “ \rangle ” when $n = 1$.

Our proof is based on the thesis that the scheduling policy at each node by which (1) a next operation is invoked as soon as the current one is suspended, and (2) operations are rescheduled in the order in which they were suspended, combined with the use of view series, does not affect the correctness of the algorithms. The thesis is formalized as the following theorem.

Theorem 3 For any series of outstanding requests $\langle o_1, \dots, o_n \rangle$ ($n \geq 2$) and authoritative state S_0

$$\sigma(S_0, \langle o_1, \dots, o_n \rangle) = \sigma(\sigma(S_0, \langle o_1, \dots, o_{n-1} \rangle), o_n)$$

Proof: For simplicity, we ignore the situation that a delete is requested for an address that is not stored, or an insert for an address that is already stored. The proof is by induction, starting with the case that $n = 2$.

Case 1: o_1 is a `delete_addr` request. A view that is appended by the delete operation to the record's present view series is always unconditionally applied upon the completion of the operation. If no view is appended, the delete request is forwarded to the parent without affecting the record's present state. In other words, $\sigma(S_0, o_1) = \sigma_{pre}(S_0, o_1)$. Consequently,

$$\sigma(S_0, \langle o_1, o_2 \rangle) \stackrel{def}{=} \sigma(\sigma_{pre}(S_0, o_1), o_2) = \sigma(\sigma(S_0, o_1), o_2)$$

Case 2: o_1 is an `insert_chk` request. Let a be the address that is to be checked by the operation `insert_chk`.

If a is to be stored at node N , then the state viewed by o_2 after o_1 has been preprocessed is the same as $\sigma(S_0, o_1)$, i.e. $\sigma(S_0, o_1) = \sigma_{pre}(S_0, o_1)$ so that we again have $\sigma(S_0, \langle o_1, o_2 \rangle) = \sigma(\sigma(S_0, o_1), o_2)$.

Assume that a is eventually to be stored in another node M . If M is at a lower level than N , it is not hard to see that o_1 will append a view in which a forwarding pointer is set. This view will later be applied as all higher level nodes will recursively install a pointer as well, and return `OK`. Consequently, $\sigma(S_0, o_1) = \sigma_{pre}(S_0, o_1)$ and thus $\sigma(S_0, \langle o_1, o_2 \rangle) = \sigma(\sigma(S_0, o_1), o_2)$.

Now assume that M is at a higher level than N , and that o_1 appended a view which will later be removed. We now have that $S_0 = \sigma(S_0, o_1)$, but $\sigma(S_0, o_1) \neq \sigma_{pre}(S_0, o_1)$. Three situations can occur:

- If o_2 is an `insert_addr` request for address b , o_2 will invoke `insert_chk` at the parent, just as o_1 did. The parent will eventually return a `DELETE` for o_1 as well as o_2 , leaving the contact record in state S_0 . In other words, $\sigma(S_0, \langle o_1, o_2 \rangle) = S_0$.
Because b will also be stored in M , invoking o_2 after o_1 has completed will not lead to a change of state, i.e. $\sigma(\sigma(S_0, o_1), o_2) = \sigma(S_0, o_2) = S_0$, so that

$$\sigma(S_0, \langle o_1, o_2 \rangle) = S_0 = \sigma(\sigma(S_0, o_1), o_2)$$

- If o_2 is an `insert_chk` request for address b , o_2 will follow exactly the same flow of control as for o_1 , appending a view that will later be removed, and leaving the contact record in its initial state $\sigma(S_0, o_1) = S_0$. In other words, $\sigma(S_0, \langle o_1, o_2 \rangle) = S_0$.
Because o_2 will behave exactly the same as o_1 , o_2 will leave its initial state unaffected, i.e. $\sigma(\sigma(S_0, o_1), o_2) = \sigma(S_0, o_1) = S_0$, so that we again obtain

$$\sigma(S_0, \langle o_1, o_2 \rangle) = S_0 = \sigma(\sigma(S_0, o_1), o_2)$$

- Finally, let o_2 be a `delete_addr` request for address b . It is easy to see that if $a = b$, `delete_addr` will undo the effects of o_1 by appending an appropriate view at N . That view is unconditionally applied, bringing the contact record back into state S_0 . As before, we have that $\sigma(S_0, \langle o_1, o_2 \rangle) = S_0$.

If o_1 had already been completed, the contact record would be in state $\sigma(S_0, o_1) = S_0$. Because the address is not stored at \mathbb{N} , o_2 will immediately forward the delete request to the parent, without even appending a view. In other words, $\sigma(S_0, o_2) = S_0$. Consequently, we have that

$$\sigma(S_0, \langle o_1, o_2 \rangle) = S_0 = \sigma(\sigma(S_0, o_1), o_2)$$

If $a \neq b$, o_1 and o_2 are unrelated and the delete request is immediately forwarded to the parent leaving the state unaffected, i.e. $\sigma(S_0, o_1) = \sigma_{pre}(S_0, o_1) = S_0$ and $\sigma(\sigma(S_0, o_1), o_2) = \sigma(S_0, o_2) = \sigma_{pre}(S_0, o_2) = S_0$. Again, we see that

$$\sigma(S_0, \langle o_1, o_2 \rangle) \stackrel{def}{=} \sigma(\sigma_{pre}(S_0, o_1), o_2) = S_0 = \sigma(\sigma(S_0, o_1), o_2)$$

Case 3: o_1 is an insert_addr request. This is completely analogous to the second case.

Now assume that the theorem holds for $n = k \geq 2$. Denote by $S_k^* = \sigma(S_0, \langle o_1, \dots, o_k \rangle)$ and $S_0^* = S_0$. By repeatedly applying the induction hypothesis, we have that $\sigma(S_k^*, o_{k+1}) = \sigma(S_1^*, \langle o_2, \dots, o_{k+1} \rangle)$. In those cases that $\sigma_{pre}(S_0, o_1) = \sigma(S_0, o_1)$, i.e. the tentative state after preprocessing o_1 is the same as the one after its completion, we have

$$\begin{aligned} \sigma(S_1^*, \langle o_2, \dots, o_{k+1} \rangle) &\stackrel{def}{=} \sigma(\sigma(S_0, o_1), \langle o_2, \dots, o_{k+1} \rangle) = \\ &\sigma(\sigma_{pre}(S_0, o_1), \langle o_2, \dots, o_{k+1} \rangle) \stackrel{def}{=} \sigma(S_0, \langle o_1, \dots, o_{k+1} \rangle) \end{aligned}$$

Now consider those cases where the tentative state after preprocessing o_1 is different from S_1^* . This means that a view appended by o_1 will later be removed. This can happen only when o_1 adds either an address or a forwarding pointer, which is later removed because the (associated) address is to be stored at one of the parent nodes, say \mathbb{P} . Assume that \mathbb{N} stores all addresses for the region \mathbb{R} from which the operations o_1, \dots, o_{k+1} have been initiated.

If the regional record at \mathbb{P} for \mathbb{R} never becomes empty while processing the series $\langle o_1, \dots, o_k \rangle$, then no addresses or pointers will ever be stored authoritatively in the contact record at \mathbb{N} . This means that S_0 at \mathbb{N} will remain unaffected after having postprocessed all requests. In other words, for all k we have that $S_0 = \sigma(S_0, \langle o_1, \dots, o_{k+1} \rangle)$ so that, in particular

$$\sigma(S_0, \langle o_1, \dots, o_{k+1} \rangle) = \sigma(\sigma(S_0, \langle o_1, \dots, o_k \rangle), o_{k+1})$$

On the other hand, it may well be that after having processed the subseries $\langle o_1, \dots, o_i \rangle$ for some $i \leq k$ at \mathbb{P} , the regional record at \mathbb{P} for \mathbb{R} becomes empty. If in the meantime, shouldstore at \mathbb{P} has changed so that it now returns false, addresses may now possibly have to be inserted at lower level nodes. In particular, \mathbb{N} may now be permitted to store addresses. However, we now also have $\sigma_{pre}(S_0, \langle o_1, \dots, o_i \rangle) = S_0$ so that

$$\begin{aligned} \sigma(S_0, \langle o_1, \dots, o_{k+1} \rangle) &= \sigma(\sigma_{pre}(S_0, \langle o_1, \dots, o_i \rangle), \langle o_{i+1}, \dots, o_{k+1} \rangle) && \text{(by definition)} \\ &= \sigma(S_0, \langle o_{i+1}, \dots, o_{k+1} \rangle) \\ &= \sigma(\sigma(S_0, \langle o_{i+1}, \dots, o_k \rangle), o_{k+1}) && \text{(by induction)} \\ &= \sigma(\sigma(\sigma_{pre}(S_0, \langle o_1, \dots, o_i \rangle), \langle o_{i+1}, \dots, o_k \rangle), o_{k+1}) \\ &= \sigma(\sigma(S_0, \langle o_1, \dots, o_k \rangle), o_{k+1}) && \text{(by definition)} \end{aligned}$$

We can now conclude that in all cases $\sigma(S_k^*, o_{k+1}) = \sigma(S_0, \langle o_1, \dots, o_{k+1} \rangle)$, completing our proof by induction on k . ■