Efficient Tracking of Mobile Objects in Globe

ALINE BAGGIO, GERCO BALLINTIJN, MAARTEN VAN STEEN AND ANDREW S. TANENBAUM

Department Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
Email: steen@cs.vu.nl

A location service tracks and locates objects. Such a service should provide efficient means for updating and looking up an object's address, especially for those that are mobile. However, current location services have limited scalability due to poor exploitation of locality and ineffective caching. An important aspect of efficient caching in the presence of mobility is to identify boundaries of the region within which a mobile object usually remains. Caching a reference to such a region rather than to the object itself ensures that the cached entry remains stable. Identifying a region requires dynamically taking migration patterns into account. This paper describes a scalable location service that efficiently supports tracking mobile objects, partly by dynamically adapting to the mobile behavior of each object separately.

Keywords: mobile objects, location service, location caching, wide-area distributed systems

Received 23 November 2000; revised 21 June 2001

1. INTRODUCTION

With the increasing mobility of objects in the Internet, such as users, hardware and software resources, efficiently tracking those objects has become critical. **Location services**, which have traditionally been part of phone systems, perform the task of tracking objects. Many ways exist to implement location services [1], but matters become complex when dealing with a large number of objects spread across a wide-area network. As part of the Globe project we are building a location service designed to handle a vast number of potentially mobile objects distributed worldwide. The Globe location service is built as a distributed search tree, representing a partitioning of the underlying network [2].

To designate objects, the Globe location service uses universally unique identifiers called **object handles**. An object handle is a pure name: it contains no information or hints on how and where to locate the designated object [3, 4]. An object's location, in turn, is described by means of a **contact address**, which contains information on where and how to contact an object. An object handle maps to possibly several contact addresses, for example, if the object is replicated. Each node in the tree of the Globe location service maintains information about objects. A node supports lookup operations by which an object handle is mapped to a contact address, and update operations by which a contact address is added or removed.

To achieve scalability, exploiting locality is extremely important. A request to lookup or update an object handle should therefore preferably visit nodes close to the client that initiated the operation. Furthermore, the number of nodes visited should be minimal, even in the presence of objects with a high migration frequency. To improve scalability of lookup operations, research suggests caching results [5, 6] in addition to applying other techniques, such as distribution and replication [7]. However, caching is effective only when there is a stable name-to-address mapping, which is not the case for highly mobile objects.

To overcome the limitations of result caching, we have devised a **location caching** scheme in which we cache a reference to a node holding an address of the object, rather than caching the address of the object itself. With a location cache, a lookup operation is done at best in two hops: one to get the location of an address from the cache, the other one to retrieve the current address from that location.

This paper presents the design of a location caching scheme for the Globe location service along with a detailed description on how to efficiently lookup a contact address for an object. The paper supplements [8] and together they provide the algorithmic details of a worldwide scalable location service. The paper is organized as follows. Section 2 gives an overview of the Globe location service. Section 3 presents the basic mechanisms for effective caching, followed by a detailed description of the lookup operation in Section 4. Section 5 describes a simple interface to the location service for supporting mobile objects. Section 6 discusses our work in relation to other research and Section 7 draws our conclusions.

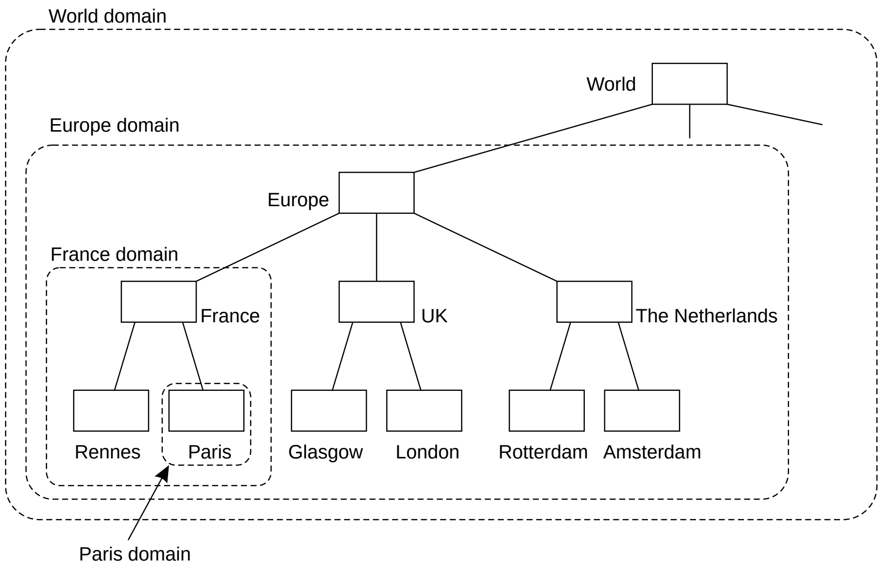

FIGURE 1. The organization of the Globe location service into domains.

2. THE GLOBE LOCATION SERVICE

The Globe location service is built as a hierarchy of domains. It supports basic operations to look up contact addresses, and to insert and delete them. Insertion and deletion are jointly referred to as update operations. This section outlines the overall organization of the Globe location service as well as the approach for looking up and updating contact addresses.

2.1. General organization

The location service is based on a hierarchical organization of the network into **domains**, similar to the organization of domains in the Domain Name System (DNS) [9]. Domains divide the underlying network into geographical, administrative, or network-topological areas. For example, a lowest-level domain (called a **leaf domain**) may represent the network of a city, whereas the next higher-level domain represents the country or state in which this city is located. The highest-level domain represents the entire network, such as the Internet. This organization is shown in Figure 1. For example, the domain *Europe* consists of three subdomains: *France*, *UK* and *The Netherlands*. Domain *France* is subsequently divided into two subdomains: *Rennes* and *Paris*. A further division into lower-level domains could represent a neighborhood or a campuswide network of a university.

Within the Globe location service, a domain is represented by a **directory node**. Each node is responsible for keeping track of all objects located in its own domain. As a consequence, the root node knows about all objects worldwide and can locate any of them if need be. A directory node uses a separate **contact record** for each object registered in its domain. A contact record consists of a number of **contact fields**, one for each subdomain (see Figure 2). A contact field stores one or more addresses for its associated subdomain where the object can be contacted. Alternatively, it stores a **forwarding pointer** which indicates that an address is stored at a lower-level node in its subdomain. If a contact field for a given subdomain is empty, the object has no contact address in this subdomain.

Figure 2 shows an example of a tree in which a replicated object is available at two locations: in *Paris* and in *London*. For each contact address, there is a chain of forwarding pointers starting from the root node (labeled World) to the node where the address is actually stored. By default, an address is stored in a leaf node. However, it is possible to store an address at an intermediate node, such as in node UK, which stores the *London* address.

To avoid having the root node and other high-level nodes become a bottleneck, we partition a directory node into several physical nodes (i.e., servers running on separate machines). Partitioning is transparent: the physical nodes form together a logical directory node. Each physical node is responsible for a subset of objects residing in the domain associated with the logical node. Partitioning is done by means of a location-aware hashing technique of which the details are described in [10].

2.2. Lookup operations

The basic approach to looking up an address in the Globe location service is fairly simple. However, it may lead to a waste of resources and exhibit unscalable behavior. In this section, we first present the basic approach for looking up an address and then show how to overcome its problems. We discuss the role of caching and how to make it effective in the case of mobile objects.

2.2.1. Basic approach

In Figure 2, consider a client residing in leaf domain *Amsterdam* wishing to look up a contact address for an object replicated in *Paris* and *London*. This client sends a lookup request to node Amsterdam, which checks whether the object is located in its own domain. If the object is unknown to node Amsterdam, it forwards the request to its parent node

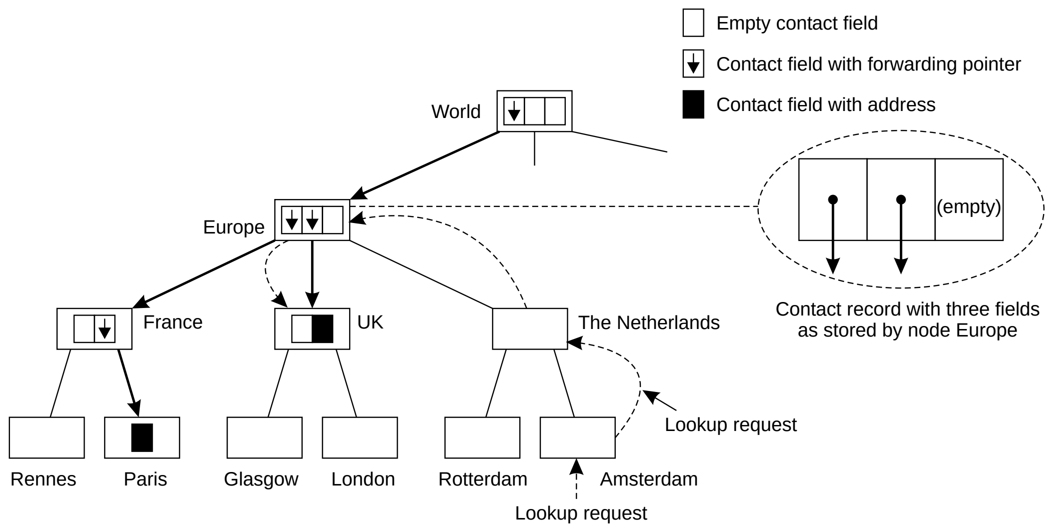

FIGURE 2. The organization of the Globe location service for a single replicated object.

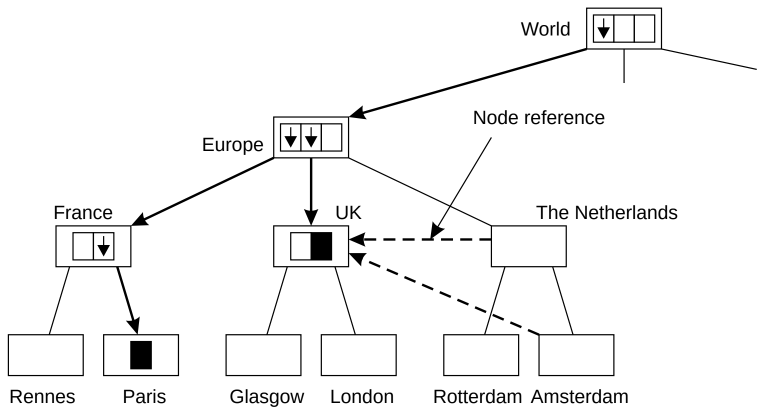

FIGURE 3. The principle of location caching.

The Netherlands. Forwarding continues until a node with a contact record for the object is reached, which in the worst case is the root. In our example in Figure 2, the lookup request is forwarded from node Amsterdam up to node Europe. At this point, it is known that the object can be contacted in subdomains *UK* and *France*. The lookup request is forwarded to the directory node of either subdomain, in our example node UK. Once an address has been found, the reply follows the reverse path of the request, back to the requesting client.

A naive approach to selecting a subdomain for forwarding the lookup request as in the case of *Europe* is to choose either node France or node UK at random. Alternatively, the lookup request can be forwarded to all subdomains in parallel. However, this would generally incur a waste of resources. A better approach is to attach a timestamp to each contact field and use it to take a decision. The timestamp records the last time the contact field became nonempty. This time indicates when an object moved into the subdomain. The lower the timestamp, the longer the object has been continuously residing in the subdomain. Following the least-recent forwarding pointer leads to a subdomain where the object has been available the longest. In our location service, we generally prefer to select the most stable address we can find. Following the least-recent forwarding pointer is based on the expectation that the address that is found in the associated subdomain will be more stable than an address found by following a more recent forwarding pointer.

Occasionally, a lookup operation may reach a node that stores both an address and a forwarding pointer. If so, the lookup operation is not forwarded but returns the address found. We consider it better to return an address as soon as possible than to continue searching. Such a strategy reduces the length of the search path instead of the one that returns an address from the most stable location.

2.2.2. Role of caching

In the basic approach to looking up an address we may have to go all the way up to the root and then back down the tree via a path of forwarding pointers. Having such a global tree traversal is generally inefficient as is also demonstrated for DNS [11]. We need to make sure that this rarely happens. Efficiency is considerably improved by caching results from previous lookup operations. Unfortunately, caching a contact address of a mobile object is not effective as we can expect such an address to rapidly become invalid. Caching is effective only when the original data do not change often.

Returning to Figure 2, consider a mobile object that migrates within domain *France*. Regardless of the object's current address, node France always stores a forwarding pointer to one of its child nodes. By caching a reference to node France instead of the object's current address, we quickly locate the object, no matter how often it migrates within domain *France*.

To further improve performance, a contact address can be stored at an intermediate node. The current address of our mobile object would therefore be stored in node France. This happend, for example, at node UK, which thus became a **stable address location**, also called a **stable location**. A stable location for an object is a directory node that always stores a contact address for the object. A lookup operation visiting such a node finds the address stored there. While the response to the lookup request travels back to the node that initiated the lookup, it triggers the caching of a reference to the stable location. As illustrated in Figure 3 each node on the return path (i.e., The Netherlands and Amsterdam, respectively), stores a pointer to node UK. As an effect of using location caches, a subsequent lookup request from domain *Amsterdam* for the same object visits only two nodes: Amsterdam where it finds a cached pointer to UK, and node UK where it finds a contact address.

2.3. Update operations

Figure 4 shows the steps for inserting a contact address in the location service from domain *London*. Each contact address is associated with a specific leaf domain. This leaf node is contacted when an insert request is issued and, in principle, stores the address in the contact record associated with the object. If the node does not yet have a contact record for the object, it creates one and asks its parent node permission to store the address. If permission is granted, the parent node installs a forwarding pointer. If necessary, it also creates a contact record for the object and contacts its own parent node. The insert operation stops as soon as the request reaches a node where the object is already known [which is node Europe in Figure 4 (a)]. In the worst case, the request is forwarded to the root node.

During the insertion process, a parent node has the right to store the address itself instead of a forwarding pointer to the requesting child node. This is the case for node UK in Figure 4 (b). The child node (London) is not granted permission to store the address and no forwarding pointer is installed in the parent node (UK). Instead, the parent stores the address.

Deleting an address is straightforward. A deletion request is sent to the node of the address's leaf domain. It is forwarded upwards until the contact address is found where it is deleted from the contact record. If the deletion leaves a contact record empty, the record is deleted as well. The parent node is then requested to remove its forwarding pointer, which may, in turn, lead to the further deletion of forwarding pointers and contact records at higher-level nodes. If the address is not found at all, the delete operation simply fails.

By deleting a contact record when it becomes empty, we also delete its timing information. This may seem a rather crude decision, but has the benefit of simplicity: a directory node does not keep track of objects that are currently not in its domain. An alternative is to keep the timing information even when the contact record is deleted so that it can be used when the object returns. However, this would require additional facilities, such as garbage collecting old information. It is difficult to evaluate at this point whether the benefits of keeping timing information warrant such extra data management. For this reason, we have decided to leave it out of the current design, and subject it to further research.

3. EFFECTIVE LOCATION CACHING

In the Globe location service, caching is based on the use of stable address locations. In this section, we present how to actually identify a stable location by using timing information and how to maintain this information in contact records. We also show how to adapt the stable location to the behavior of an individual object and outline the management of location caches.

3.1. Identifying a stable location

A stable location is identified by keeping track of when updates to a contact record take place. Each field f of a contact record has an attribute $T_{filled}(f)$ that records the last time an empty contact field was filled with an address or forwarding pointer. In other words, it records the last time an object migrated into a given subdomain. We use the notation $T_{filled}(f) \neq \perp$ to indicate that $T_{filled}(f)$ is defined. Migrations *within* the same subdomain are not significant for selecting a stable location and are therefore not recorded. Whenever an update operation adds an address or pointer to a previously empty field, a **history value** H is computed for the contact record CR as a whole using the following aging algorithm:

$$\begin{aligned} D &:= T_{now} - \max\{T_{filled}(f) \neq \perp \mid f \in \mathsf{CR}\} \\ H &:= \alpha \cdot D + (1-\alpha) \cdot H_{old} \end{aligned}$$

where H_{old} denotes the previous history value and T_{now} the time at which the update is taking place. The variable D represents the elapsed time (i.e., duration) since the object last moved into any subdomain it did not yet reside in. Computing the last time when such a move took place is done by looking at the value $T_{filled}(f)$ for each contact field f where the object resides, and taking the highest value. If D is low, the object recently moved to a subdomain where it did not yet offer a contact address. If D is always low, we are apparently dealing with an object that is moving frequently between subdomains.

We are interested in whether the object frequently moves between subdomains. In such cases, it may be sensible to store the object's current contact address at the node where H is computed instead of at one of the child nodes. In order not to rely on just the most recent value of D but also to take preceeding values of D into account, we compute the history value H from D and a weighted aggregated value over previous durations. The initial value for H is set to a

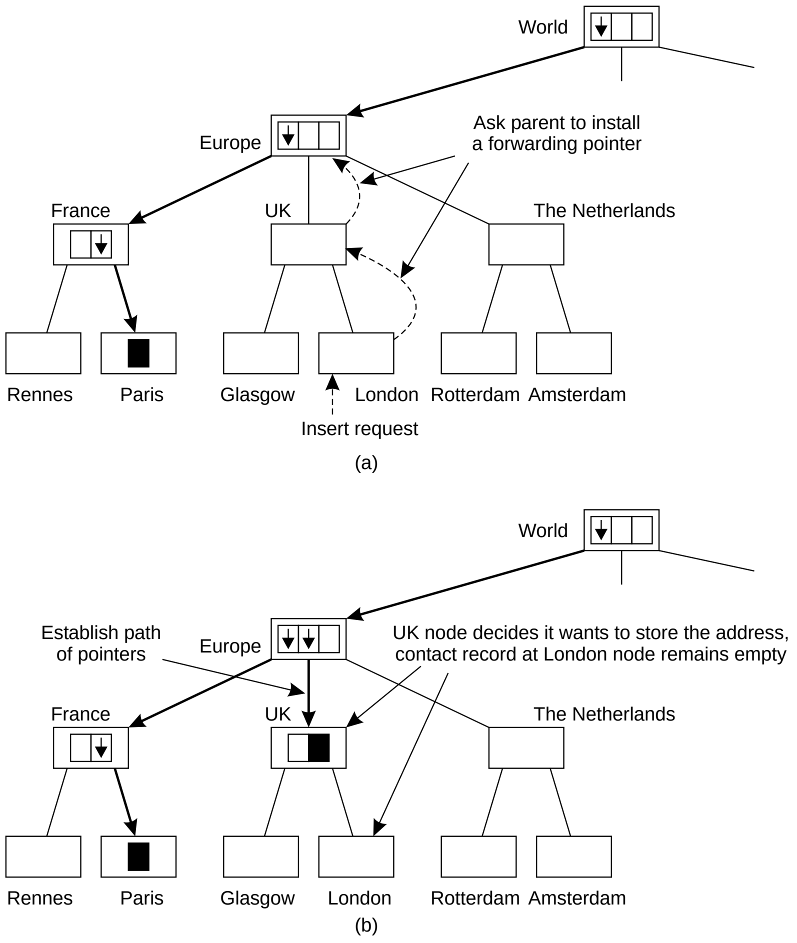

FIGURE 4. The principle of inserting an address. (a) The insert request travels upwards to the first node where the object is known. (b) A path of forwarding pointers is established.

large, finite number; for each field f, $T_{filled}(f)$ is initially undefined. $T_{filled}(f)$ also becomes undefined when the object moves out of the associated subdomain.

To illustrate, consider a mobile object with only a single address for which a given node calculates the history. If the object is frequently migrating between subdomains, then the history value is small. In addition, if the contact record is relatively old, the node containing this record is a potential stable location for the object. On the other hand, a large history value shows that it is some time ago that the object migrated into a subdomain where it did not have an address before. From the node's perspective, the object is hardly migrating between its subdomains. This still makes the node a stable location, but it is presumably not the best one.

3.2. Adapting the stable location

There are two cases to consider for changing a stable location. First, a node may decide it is a better location to store an object's addresses than any of its child nodes. This corresponds to storing addresses higher in the tree, also called **upwards**. Second, it may also occur that a child node is potentially better as a stable location. In that case, addresses need to be stored lower in the tree, that is, **downwards**.

3.2.1. *Storing an address upwards*

To move a stable location upwards, each node maintains a **mobility threshold**. This threshold is used by a node to decide if it should start storing the contact addresses of an object instead of its children. To explain, consider a node N and assume that the history value of an object drops below N's mobility threshold. Intuitively, this means that the object is moving often between the subdomains of N. In that case, it is more efficient to store the contact address of the object at N instead of regularly switching the storage location between the children of N.

Letting N store the address is relatively simple. Whenever the object migrates into a subdomain where it did not yet have an address, the child node of N associated with this subdomain eventually requests node N to install a forwarding pointer. Based on the history value, node N can then refuse to grant the request and instead store the address itself in the contact field for that subdomain.

Objects that frequently migrate within a large area impose a potential performance problem. Assume that the current address of such an object is stored by a higher-level node. As an update operation is always initiated at a leaf node, it is necessarily forwarded along each node on the path to the stable location. Nevertheless, such an update is cheaper than

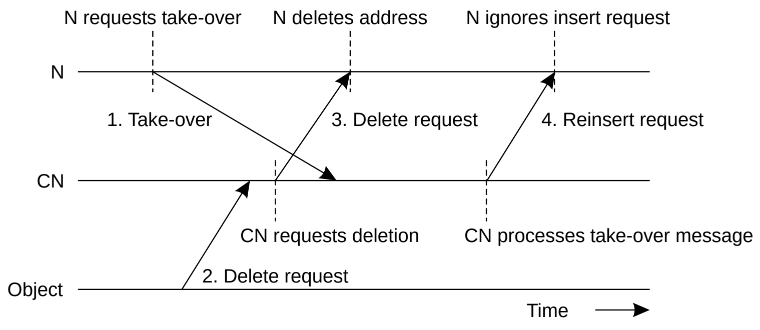

FIGURE 5. A possible race between moving an address downwards and deleting it.

keeping the current address at one of the lower-level nodes, which requires constructing a path of forwarding pointers to the new address, and removing the path to the old address.

3.2.2. Storing an address downwards

An object can settle down within a specific subdomain. This means that the object migrates only within this subdomain or does not migrate anymore. In such a case, it is better to let the child node for that subdomain store the current address. From a node's perspective, settling down means that $T_{filled}(f)$ for the field associated with that subdomain no longer changes. This situation is detected by checking whether the last time an object migrated into a subdomain exceeds a maximum duration D_{max}:

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

T_{now} denotes the time at which this operation is performed. Whenever this expression becomes true, the child node is instructed to "take over" the address. Checking the duration time is done at the node where the address is stored each time the contact record is accessed by a location service operation. Additionally, checking is done by means of a background process.

Downward storage of addresses requires an explicit action to be taken at the node where the address is stored. Only this node has a representative value of the history and is able to decide when to store an address downwards. Storing downwards requires two suboperations: (1) storing the address at the child node and (2) installing a forwarding pointer at the current node in place of the address. These two suboperations should be carried out as a single atomic operation.

To avoid the overhead of a full-blown transaction, we take the following approach. Whenever a node N decides that its child node CN should take over the storage of addresses for its subdomain, it sends a **take-over message** to CN. This message lists the addresses for which CN should request storage. Node N does not maintain any state on whether it requested one of its child nodes to take over the storage of addresses. Consequently, it may decide at any point to send out another request. Keeping nodes as ignorant as possible, in this case by forgetting that a take-over message has been sent, keeps the overall management of a node simple. In particular, each request from a child node can always be considered afresh and independent of previous requests.

When a child node receives a take-over request from its parent node, it treats it as an insert operation. Because the child node has no contact record for the associated object, it requests its parent node to install a forwarding pointer, just as it would usually do. The parent node N will notice that it already stores the addresses and can decide to replace those addresses with a single forwarding pointer to the child node CN.

This approach introduces a race condition between a deletion and an insertion of the same address. Assume parent node N stores address addr but wants child node CN to store it. Node N sends a take-over message to CN, shown as message 1 in Figure 5. Around the same time, the object associated with addr decides to delete its contact address (shown as message 2). If this deletion (message 3) takes place in N before CN decides to insert addr (message 4), the insertion by CN should fail.

This race condition can be circumvented by marking the insert request CN sends to its parent node as being triggered by one of the higher-level nodes. This makes the request explicitly marked as a *re*-insert. This is the only adaptation required for the insert operation. Following the normal insert procedure, the request is passed to the parent. If the address is no longer stored there, the parent can only conclude that it has already deleted the address (and possibly even the entire contact record for the object). Therefore, if no more information is found about the object, the re-insertion fails.

3.3. Cache management

To store pointers to stable address locations, each node is assumed to have a location cache. A cache entry represents a single object by a record maintaining two sets of locations, local and remote, as shown in Figure 6 (we use an Ada-like notation as in [8]). Each node has an associated unique node identifier of type NodeID. For a node N, the field localPtrs stores pointers to directory nodes in the same domain as N. This includes all nodes from the subtree rooted at N. In contrast, remotePtrs stores pointers to nodes outside the current domain. For example, in Figure 3, the cache for node The Netherlands would contain only a reference to node UK stored in remotePtrs, whereas the field localPtrs would be empty.

A location stored in the cache is associated with an expiration time. This time is computed by means of a function

```
type CacheEntry is
  record
    localPtrs : set of NodeID := ∅;                  -- Pointers to locations in current subdomain
    remotePtrs : set of NodeID := ∅;                 -- Pointers to locations outside current subdomain
    expirationTimes : set (NodeID) of Date;          -- Indexed set of time when a cached reference expires
  end record;

type Cache is set (ObjectHandle) of CacheEntry;      -- Indexed set of cache entries

type StoredAddress is
  record
    addr : Address := NIL;                           -- The contact address found during a lookup
    age : Date := 0;                                 -- Elapsed time since the address was stored in the location service
    node : NodeID := NIL;                            -- The node where the address was stored
  end record;
```

FIGURE 6. The data structures for caching.

```
(1) procedure cache_insert(object : ObjectHandle, storedAddr : StoredAddress) is
(2)   if storedAddr.node ∈ domain(thisNode)
(3)     then cache(object).localPtrs := cache(object).localPtrs + {storedAddr.node}
(4)     else cache(object).remotePtrs := cache(object).remotePtrs + {storedAddr.node}
(5)   end if
(6)   cache(object).expirationTimes(storedAddr.node) := expire(storedAddr.age);
(7) end cache_insert
```

FIGURE 7. Inserting a cache entry.

expire which is presently left unspecified. A simple strategy is to have the expiration time depend on the time the address was inserted at the node where it was found. The more recent its insertion time, the sooner the reference to its current location expires. This approach resembles the Alex cache replacement policy applied to Web caches [12]. This strategy can be tuned by using the history value as a way to predict changes more accurately.

A cache itself is represented as an indexed set of cache entries, in which we use an object handle as an index. This representation also requires that we parameterize operations with an object handle in order to perform cache operations.

Figure 6 also shows the data type StoredAddress, which is a convenient representation of a contact address returned as the result of a lookup request. It consists of a record with three fields: the address, the time since it was stored in the location service and the reference to the address location in the form of a node identifier. A contact address is represented by the opaque data type Address, while Date is used to represent time.

Inserting an entry into a cache is straightforward and is shown in Figure 7. Due to the idempotent nature of set operations, we need not check whether a node was already cached. We only need to check whether the node where the address was found is local, that is, whether it lies in the same domain as the current node (line 2). The identifier of the current node is available through the variable thisNode. The function call domain(thisNode) returns the node identifiers of the nodes in the subtree rooted at node thisNode. Depending on the result, the node reference is stored in the local (line 3) or in the remote (line 4) set. Finally, the expiration time for the entry is updated.

To look up a cache entry, we distinguish whether the lookup should return a local reference or a remote one, which is expressed by the strategy parameter shown in line 1 in Figure 8. A cache lookup always returns the "nearest" reference, expressed by means of a function nearest. The actual metric for nearest may vary but is by default the number of links in the tree that need to be traversed from the current node to the referenced node. As an alternative to selecting either a local or remote reference, the best reference can also be looked up. In that case, the nearest reference in the set of local *and* remote references is returned (line 4).

Cache entries are deleted when they prove to be invalid during a lookup operation. Offline purging of cache entries is also triggered regularly, resulting in removing each entry that has reached its expiration time.

4. LOOKUP OPERATIONS

To make use of the location caches, the basic lookup operation has to be extended. The lookup has to ensure that references to stable locations are cached, but also be able to get locations from the cache and retrieve addresses from them. This section first presents the data structures for contact records and then the algorithm for looking up a single contact address. Initially we make no distinction between different contact addresses. Later, we discuss a mechanism to select specific kinds of contact addresses while maintaining the principle of locality. Strategies for distinguishing replicas are independent of the caching mechanisms described earlier.

```
(1) procedure cache_lookup(object : ObjectHandle, strategy : (local, remote, best)) return NodeID is
(2)   if strategy = local then return any in nearest(cache(object).localPtrs, thisNode) end if
(3)   if strategy = remote then return any in nearest(cache(object).remotePtrs, thisNode) end if
(4)   return any in nearest(cache(object).localPtrs + cache(object).remotePtrs, thisNode)
(5) end cache_lookup
```

FIGURE 8. Looking up a cache entry.

```
type ContactField is
  record
    addrSet : set of Address := ∅;                     -- Set of contact addresses for subdomain
    isPtr : Boolean := false;              -- True iff contact field is forwarding pointer to child
    empty : automatic Boolean ≡ (addrSet = ∅ and not isPtr);  -- True iff contact field has no data
  end record;

type ContactRecord is set (NodeID) of ContactField;              -- Indexed set of contact fields
type ContactRecordDB is set (ObjectHandle) of ContactRecord;     -- Indexed set of contact records
type History is set (ObjectHandle) of float;      -- Stores the current history for each contact record
```

FIGURE 9. The data structures for a contact record, and the definition of a database of contact records.

4.1. Data structures

The definition of a contact record is shown in Figure 9. A contact field consists of a set of contact addresses addrSet. The boolean isPtr is set to true if and only if the field plays the role of a forwarding pointer to the associated subdomain. Finally, empty indicates whether or not any data is stored in the contact field. We assume that empty is set *automatically* to true whenever addrSet becomes empty and isPtr is false. Otherwise, it is automatically set to false. A contact record is represented as an indexed set of contact fields. Furthermore, we represent the database of contact records as stored at a particular node as a set indexed by object handles. In addition, we use a separate data type for history values, also represented as a set indexed by object handles.

Figure 9 shows no explicit timing information associated to contact records or contact fields. Instead, we assume that all containers (i.e., sets or records) keep an account of when elements are added or modified. For any element a in a container A,

```
date of a
```

returns the time when a was added to A or when it was last modified. For example, if for a given contact field f empty is false, **date of** empty returns the value $T_{filled}(f)$ described in Section 3, that is, it returns the last time the object moved into the subdomain associated with f.

Using this timing information, it is possible to select the oldest element in a container. For example, if cr is a contact record stored in node UK from Figure 2 then

```
addr := oldest in cr(London).addrSet
```

returns the least recently inserted address in the set of address for the *London* domain. In the case of an indexed set, the *index* of the least recently modified element is returned. In our example

```
child := oldest in cr
```

returns the node identifier of the child node whose contact field in cr has least recently been modified.

4.2. The lookup algorithm

Looking up an address in the location service without compromising scalability requires that we exploit locality as much as possible. A lookup operation is always initiated at a leaf node and gradually propagates upwards if need be.

When combining lookups with location caches, we take the following approach. Looking up an object, we first look for any contact address for the object in the current node. If no address is found, the lookup checks whether an address is available anywhere in the current domain, that is, the subtree rooted at the current node. If this also fails, a remote domain is inspected if a reference is found in the location cache. If all possibilities have been unsuccessful, the lookup request is passed to the parent node (unless the parent was actually the requesting node).

This strategy leads to the lookup algorithm as shown in Figure 10. The operation takes three parameters: caller identifies the node from which the operation has been received, object identifies the object for which an address is being looked up, and subDomOnly indicates whether it is permitted to continue searching only in subdomains.

We assume that each node has an associated contact record database, represented by a variable CRDatabase. The lookup operation starts by looking up the contact record for the given object, represented by the variable currentCR in line 3. The value of currentCR will be set to NIL if the object is not known at the current node. The pseudo code in lines 6–27 shows the case where the lookup operation has reached a node where the object is known, that is, it has found a contact record currentCR for the object.

The lookup operation then checks whether the contact record stores addresses. If so, it chooses the oldest contact field storing an address (line 8), where oldest is determined by the least recent update to that contact field's set of addresses. Then it selects the oldest address from this set (line 10) and returns it as the result of the lookup (line 11).

If no address is found in the object's contact record,

```
procedure lookup(caller : NodeID, object : ObjectHandle, subDomOnly : Boolean) return StoredAddress is
  storedAddr : StoredAddress := NIL;
  currentCR : ContactRecord := CRDatabase(object);
  -- ----------------------------------- LOCAL SEARCH -----------------------------------
  if currentCR $\neq$ NIL then
    -- We should be able to find something at this node. Check if there is a contact field containing an
    -- address and pick the oldest address that can be found. This address is probably the most stable one.
    child : NodeID := oldest in currentCR with currentCR(child).addrSet $\neq \emptyset$;
    if child $\neq$ NIL then
      addr : Address := oldest in currentCR(child).addrSet;
      return (addr, date of addr, thisNode);
    end if
    -- No address is stored in this contact record. Continue the lookup by checking the location cache.
    storedAddr := check_cache(object, local);
    if storedAddr $\neq$ NIL then  return storedAddr end if
    -- Cache lookup also failed. Continue inspecting each subdomain until an address is found.
    children : set  of NodeID := {child $\in$ index of currentCR with currentCR(child).isPtr};
    while children $\neq \emptyset$ loop
      -- Choose the child for which a forwarding pointer has been stored the longest time.
      child := oldest in currentCR with child $\in$ children;
      children := children $-$ {child};
      storedAddr := child.lookup(thisNode, object, subDomOnly);
      if storedAddr $\neq$ NIL then
        cache_insert(object, storedAddr);
        return storedAddr;
      end if
    end loop
  end if
  -- ----------------------------------- REMOTE SEARCH -----------------------------------
  -- No address has yet been found. If this lookup was initiated from inspecting a location cache first, then stop
  -- searching and give up. Otherwise, proceed with the lookup by checking the cache for a remote domain. If an address
  -- is found, return it, otherwise, continue with the parent node to broaden the search region. If that
  -- fails as well, give up.
  if not subDomOnly then
    storedAddr := check_cache(object, remote);
    if storedAddr $\neq$ NIL then  return storedAddr end if
    -- So far nothing has been found. Forward the request to the parent thus broadening the search region.
    if caller $\neq$ parent then
      storedAddr := parent.lookup(thisNode, object, false);
      if storedAddr $\neq$ NIL then
        cache_insert(object, storedAddr);
        return storedAddr;
      end if
    end if
  end if
  return NIL;
end lookup;
```

FIGURE 10. The lookup operation.

```
(1)  procedure check_cache(object : ObjectHandle, strategy : (local, remote, best)) return StoredAddress is
(2)    storedAddr : StoredAddress := NIL;
(3)    cachedNode : NodeID := cache_lookup(object, strategy);
(4)    if cachedNode $\neq$ NIL then
(5)      storedAddr := cachedNode.lookup(thisNode, object, true);
(6)      if storedAddr $\neq$ NIL then cache_insert(object, storedAddr) end if
(7)      if storedAddr = NIL or  else storedAddr.node $\neq$ cachedNode then
(8)        cache_delete(object, cachedNode)
(9)      end if
(10)   end if
(11)   return storedAddr;
(12) end check_cache;
```

FIGURE 11. Operation for checking the cache.

the location cache is inspected by calling the procedure check_cache (Figure 11), which is discussed below. Checking the cache may imply doing a lookup in one of the subdomains. If an address is found at this point, it always resides in the current domain, which is specified by the strategy local in line 14. Any address found is returned, successfully completing the lookup operation (line 15).

If the cache lookup fails, the lookup continues by inspecting each subdomain for which the current contact record stores a forwarding pointer. For this purpose, we construct the set children in line 17 from the index set of currentCR. Note that the caller is never in this set. If this were the case, then the node where the lookup operation is currently being executed would have a pointer to the caller, meaning that an address is stored in the caller's domain. Consequently, an address would have already been found locally to the caller.

The subtree with the least recently added forwarding pointer is inspected first. The associated child node is selected and removed from the set (line 20) and its subtree is explored. Normally, this operation should succeed and return an address. If so, the address location is stored in the cache (line 24). However, in the face of a concurrent delete operation, it may be necessary to continue the search in another subtree (lines 18–27).

This process carries on until all subtrees have been inspected. At this point, there are two alternatives (line 34). First, this lookup has been invoked by a node outside the current subtree. This happens when the lookup request comes from the parent or when it has been forwarded through a node reference cached at the calling node. In the latter case, the lookup should stop and return control to the invoking node (line 46). Returning control guarantees the termination of the lookup operation but also respects the locality principle. By this principle, following a node reference implies that an address should be found either from the referenced node or from one of its subdomains. The referenced node is not allowed to contact its parent node and have the lookup request travel upwards in the tree.

The lookup request can come from the parent only if the parent had a forwarding pointer to the current node. If execution comes to line 34, this means that the address associated with that pointer has been deleted during the lookup. In this case, we allow the lookup procedure to continue searching for remote references as we explain for the second alternative.

The second alternative is that the lookup has been invoked from a leaf node in the current domain. In that case, the lookup continues by checking for a remote reference in the cache. This is expressed in line 35 by passing the parameter value remote to check_cache. If a reference is found in the cache as well as a valid address in the remote subtree, we return this address. If no address was found, the only alternative left is to forward the operation to the parent node provided it did not originally invoke the current lookup (line 39).

To complete our description of the lookup algorithm, the procedure check_cache is shown in Figure 11. It starts with looking for a cache entry (line 3), taking into account whether only local references can be returned or not. If a node is found, check_cache invokes the lookup operation at this node setting sudDomOnly to **true** (line 5), and inserts the result in the cache if the lookup was successful (line 7). If an address was found, it should normally still be stored at the same node that was referenced in the cache. If this is not the case, the cached reference is no longer valid. Apparently, an address has been stored at a lower-level node in the subtree rooted at the node for which a reference was cached. Consequently, the cached reference is removed from the cache (line 7). Note that this unconditional removal may actually be part of a *replacement* by means of the insertion of an up-to-date reference in line 6. In other words, if we found an address at node N, we first cache a reference to N in line 6. However, N may be the *new* location of that address, for which reason we remove the cached reference to its former location in line 8.

4.3. Distinguishing replicas

The lookup procedure supports replicated objects. In some cases a client does not want just any address of a replicated object, but requires one that meets its specific requirements. For example, when dealing a primary/backup replication scheme, a client may want to connect only to the primary to possibly install another backup. If such an address is available, it should be advertised and it should be possible to explicitly look it up.

One solution is that the object makes itself known under

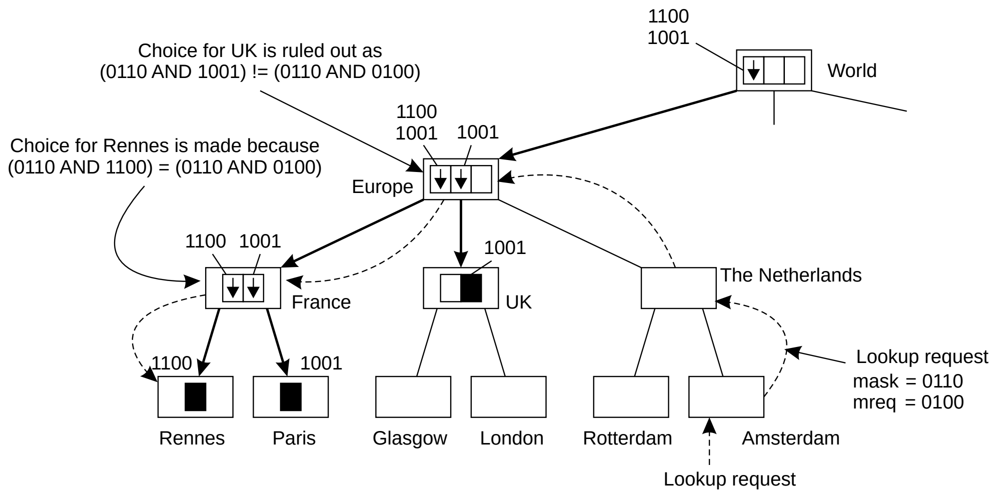

FIGURE 12. A lookup request excluding certain contact addresses using a property map.

different object handles, each one being related to a class of contact addresses. However, having objects with multiple identifiers is an undesirable feature. A client that can support multiple protocols would be forced to initiate several lookups for the same object, each lookup using a different identifier to find the nearest address. We find that having to initiate multiple lookups is an unacceptable solution.

Another solution is to extend the location service to support *(attribute, value)*–pairs and associate them with each contact address. Instead of providing only an object identifier, a client would also provide a predicate, in terms of attribute values, to be matched against the information stored along with a contact address. This approach has a serious drawback in that it violates the locality principle. At present, the location service searches for an address using only the associated object identifier. If we were to store attributes along with contact addresses, we would be forced to inspect each address and match its attribute values against the client's predicate. A lookup request would then always have to travel to the node where an address is stored and possibly return empty-handed if that address did not meet the client's requirements.

Locality requires an efficient solution by which the location service can discard a branch in the tree even if it knows that it can find an address in that branch. It should discard a branch whenever it knows the address is not what the client is looking for. In theory, arbitrary *(attribute, value)*–pairs can be stored along with forwarding pointers instead of contact addresses. However, this would introduce additional complexity and require a costly attribute-comparison operation when doing a lookup at a particular node.

Our solution is to use property maps. In our current implementation, a **property map** is a bit string that is associated with a contact address for a given object. Each bit represents an object-specific property. When set to 1, the address has that property; when set to 0, it does not. When an address is stored in the location service, its property map is stored as well. For any node N, if an address in a child node's domain has property map *m*, a copy of *m* is stored at N along with the forwarding pointer. Only if another address in the child node's domain has a *different* property map m', then node N will store m' as well.

When a client issues a lookup request, it provides a map *mreq* of required properties along with a mask *mask.* The client sets a bit in *mreq* if it wants the address to have the associated property and 0 if not. The client sets a bit in *mask* if it finds the property relevant (whether it actually wants it or not), and 0 if it has no interest in this property. A contact field is eligible for selection during a lookup operation only if it has a property map *m* stored, such that:

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

Consider the example in Figure 12 using 4-bit property maps. A client in domain *Amsterdam* is looking for an address having property 2 but not property 3. It is not interested in properties 1 and 4. Therefore, it sends a mask $mask = 0110$ along with a map $mreq = 0100$. The lookup request is forwarded upwards until a node is reached that contains information on an address matching ?10?, where "?" indicates a *don't care* value for that property. As it turns out, only the address stored in the *Rennes* domain matches these requirements, for which reason the lookup is forwarded only to this domain.

The operations for updating and looking up addresses require minor modifications to support property maps. When caching a reference to a node for a given object, we not only store the property map of the address found but the property maps of all the addresses of the associated object available at this node. A property map is deleted from a cache entry when the referred contact record does not contain a matching address anymore.

5. OPERATIONS FOR MIGRATING AN OBJECT

Besides the basic operation for looking up an address, the location service offers a convenient interface to track mobile objects. We distinguish two types of mobile objects. The first type consists of objects that disconnect while migrating from one place to another, such as a notebook. The

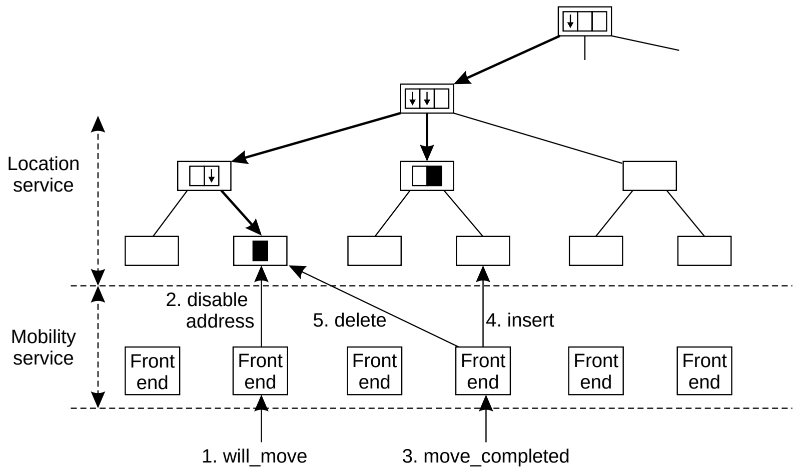

FIGURE 13. The organization of a separate layer to support mobility operations.

other type are mobile objects that can still be contacted even while they are migrating to another location. Many wireless devices but also software agents fall into this category. Replicated mobile objects are supported as well. In this case, mobility means that a specific replica migrates from one location to another, such as a local copy of a replicated file on a person's laptop.

Dealing efficiently and effectively with mobility in the location service requires that the insertion of a new contact address for a given object precedes the deletion of the old address. This ordering is necessary to allow the directory node of the domain in which the migration takes place to maintain a contact record on the object. As an example, consider a mobile object in Figure 4 with only a single address within the *UK* domain that migrates from *London* to *Glasgow*. If the object would first delete its current address, the contact record in node UK would become empty and be deleted. All information gathered on the object in domain *UK* would be lost. This loss is avoided if the new address in the *Glasgow* domain is inserted first. The contact record in node UK is preserved and still contains data when the removal of the *London* address take place.

We hide such matters behind an interface implemented as a front end to the location service as shown in Figure 13. This interface supports both the mobility of objects operating in connected mode as well as those operating in disconnected mode, while taking care that information on object migration is preserved.

The interface consists of two operations, will_move and move_completed. The operation will_move is called by an object prior to operating in *disconnected* mode at its current location. It **disables** the object's address, thus preventing clients to use the address. Disabling an address does not remove the address but records it as being invalid. A lookup operation never returns a disabled address.

When the object reconnects, it invokes the operation move_completed which takes the old and new address as input. The new address is inserted into the location service. Upon completion, the front end at the destination sends a request for deletion of the old address. If the insertion takes too long, for example because of a link failure, the deletion of the old address is sent without waiting for the insertion to complete.

For an object operating in *connected* mode, only the operation move_completed is called when the object arrives at its destination.

As an example, when an object moves in disconnected mode, it first calls will_move, as shown as step 1 in Figure 13. The front end will then request the location service to disable the object's address (step 2). When the object reconnects, it calls move_completed (step 3), resulting in the front end at the new location to first insert the new address (step 4), and delete the old address (step 5).

6. DISCUSSION AND RELATED WORK

Many experiments have shown that location-based names are not sufficient for locating mobile hosts or objects [13, 14, 15, 16, 17, 18, 19], even if their mobility rate is quite low. There are three common architectures of distributed location services.

First, a two-tier scheme approach uses home databases located in a predefined network zone. Each mobile entity is assigned both a network zone and a home which becomes permanently responsible for the mobile entity. This home database is in charge of keeping the current location of the mobile entity up-to-date and handles location requests. This approach has been used for example with GSM [20] and Mobile IP [16].

The second approach is the tree-structured hierarchical scheme. In this scheme, the network is subdivided into domains that are aggregated into larger, nonoverlapping domains. Each domain is represented by a node in the tree. The root node represents the entire network. This approach has been long used in traditional (wireline) telephony. A disadvantage of this approach compared to nonhierarchical solutions is that lookup requests may need to travel across several nodes.

A somewhat analogous approach that is used in metropolitan ad-hoc networks is the Grid location service [21]. In

Grid, the network is hierarchically organized on a per-object basis. Each host can act as a location server for an object. The goal is to find location servers that have information about the current address of an object. Servers are found using geographic routing based on the object's identifier.

The third alternative is formed by nontree hierarchies. Such an hierarchy is based on a graph-theoretical approach, as proposed in [13]. It uses regional databases to favor local operations (locate nearby entities, move to nearby locations). The hierarchy is constructed such that the maximal network distance between two sites, called diameter, is below a given upper bound. It guarantees that communication overhead for locating or moving entities are polylogarithmic in the size (number of sites) and diameter of the network.

Supplementary to these solutions is to use location caches. The solution proposed in [22] aims at reducing network traffic when locating mobile entities by using shortcut links in the form of *bypass pointers*. A bypass pointer is a direct link between two nodes in different subtrees of a search tree. Whenever such a link is found for a specific object, the lookup operation is forwarded along that link, thereby avoiding the traversal of the least common ancestor of the two nodes. Caching is therefore used to reduce the length of the path of forwarding pointers to be followed by a search request. In the Globe location service, caching is made more accurate by storing a reference to the node where the address is actually kept, reducing the length of the path to, at best, two hops.

In most other location services, the result of a lookup operation is cached, namely the address of the object that was searched for. Data caches are used only when the lookup-to-mobility ratio for a specific client is high enough [1]. In the Globe location service, data caches are currently not supported. Addresses of mobile objects are expected to be highly unstable and thus not worth caching. Moreover, stale cache entries would imply extra load on both the client and the location service. For example, the only way the location service can check the validity of a cached contact address, is to initiate a lookup to see if the address is actually stored in one of its nodes (and not only in caches). Therefore, it seems better to let a client check whether an address is still valid by contacting the object. If that fails, the client should do another lookup, indicating that the previously returned address is (apparently) not valid. Clearly, these are not viable solutions.

A useful data caching strategy would be to cache relatively stable addresses. The stability can be derived from the time the address was inserted into the location service. This ensures that the object handle-to-address mapping is stable enough to be cached. To give full control to the location service, objects must give guarantees on how long they can be contacted at that address. With a lease-based invalidation policy, the location service will be able to purge expired data from a cache without relying on clients (see also [23, 24, 25]).

7. CONCLUSION

The purpose of this study is to provide an efficient location service for both mobile and nonmobile objects. We consider a distributed search tree, whose nodes store addresses. Optimizing such a location service requires that we reduce the path for looking up addresses. Making use of what we call stable locations helps to achieve this goal.

In the Globe location service, it is relatively simple to identify stable locations. The location service collects information about object migrations. Whenever required, it ensures that the object's address is stored at the appropriate location. A stable address location strongly depends on the object's migration pattern. Each time the migration pattern changes, the location service re-evaluates the stable location. If necessary, a new one is found and the address is transferred to this node.

This paper shows how locating objects and caching results in location services can take place in an effective and efficient manner despite that objects may be highly mobile. The result of our caching policy, in combination with exploiting locality during lookup and update operations, is that searching for the current location of a mobile object is done in a scalable manner. To substantiate our claims, we are currently setting up a large-scale experiment in which the service will be used for a worldwide system for distributing software [26]. As for further research, we intend to investigate the usefulness of combining location and data caches.

REFERENCES

[1] E. Pitoura and G. Samaras. "Locating Objects in Mobile Computing." *IEEE Trans. Know. Data Eng.*, 13, 2001.

[2] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, 36(1):104–109, Jan. 1998b.

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

[4] 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.

[5] 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.

[6] B. Lampson. "Designing a Global Name Service." In *Proc. Fourth Symp. on Principles of Distributed Computing*, pp. 1–10, Minaki, Ontario, 1986. ACM.

[7] B. Neuman. "Scale in Distributed Systems." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[8] M. van Steen, F. Hauck, G. Ballintijn, and A. Tanenbaum. "Algorithmic Design of the Globe Wide-Area Location Service." *Comp. J.*, 41(5):297–310, 1998a.

[9] P. Mockapetris. "Domain Names - Concepts and Facilities." RFC 1034, Nov. 1987.

[10] G. Ballintijn, M. van Steen, and A. Tanenbaum. "Exploiting Location Awareness for Scalable Location-Independent Object IDs." In *Proc. Fifth ASCI Ann. Conf.*, pp. 321–328, Heijen, The Netherlands, June 1999. ASCI.

[11] E. Cohen and H. Kaplan. "Proactive Caching of DNS Records: Addressing a Performance Bottleneck." In *Proc. First Symp. Applications and the Internet*, San Diego, CA, Jan. 2001. IEEE.

[12] V. Cate. "Alex – A Global File System." In *Proc. File Systems Workshop*, pp. 1–11, Ann Harbor, MI, May 1992. USENIX.

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

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

[15] J. Jannink, D. Lam, N. Shivakumar, J. Widom, and D. Cox. "Efficient and Flexible Location Management Techniques for Wireless Communication Systems." *ACM/Baltzer Wireless Networks*, 3(5):361–374, Oct. 1997.

[16] C. Perkins. *Mobile IP: Design Principles and Practice*. Addison-Wesley, Reading, MA, 1997.

[17] 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.

[18] Y.-B. Lin and W.-N. Tsai. "Location Tracking with Distributed HLRs and Pointer Forwarding." *IEEE Trans. Veh. Techn.*, 47(1):58–64, Jan. 1998.

[19] S. Mohan and R. Jain. "Two User Location Strategies for Personal Communication Services." *IEEE Pers. Commun.*, 1(1):42–50, Jan. 1994.

[20] M. Rahnema. "Overview of the GSM System and Protocol Architecture." *IEEE Commun. Mag.*, 31(4):92–100, Apr. 1993.

[21] J. Li, J. Jannotti, D. D. Couto, D. Krager, and R. Morris. "A Scalable Location Service for Geographic Ad Hoc Routing." In *Proc. Sixth Int'l Conf. on Mobile Computing and Networking*, Boston, MA, Aug. 2000. ACM.

[22] R. Jain. "Reducing Traffic Impacts of PCS using Hierarchical User Location Databases." In *Proc. Int'l Conf. Communications*. IEEE, 1996.

[23] C. Gray and D. Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." In *Proc. 12th Symp. Operating System Principles*, pp. 202–210, Litchfield Park, AZ, Dec. 1989. ACM.

[24] P. Cao and C. Liu. "Maintaining Strong Cache Consistency in the World Wide Web." *IEEE Trans. Comp.*, 47(4):445–457, Apr. 1998.

[25] V. Duvvuri, P. Shenoy, and R. Tewari. "Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web." In *Proc. 19th INFOCOM Conf.*, pp. 834–843, Tel Aviv, Israel, Mar. 2000. IEEE.

[26] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. Tanenbaum. "The Globe Distribution Network." In *Proc. Ann. Techn. Conf. (FREENIX Track)*, pp. 141–152, San Diego, CA, June 2000. USENIX.