DISTRIBUTED LOCATION DATABASES FOR TRACKING HIGHLY MOBILE OBJECTS

E. PITURA and I. FUDOS

20-99

Preprint no. 20-99/1999

Department of Computer Science
University of Ioannina
451 10 Ioannina, Greece

Distributed Location Databases for Tracking Highly Mobile Objects*

Evaggelia Pitoura and Ioannis Fudos
Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
email: {pitoura,fudos}@cs.uoi.gr

Abstract

In current distributed systems, the notion of mobility is emerging in many forms and applications. Increasingly many users are not tied to a fixed access point but instead use mobile hardware such as dial-up services or wireless communications to access data independent of their physical location. Furthermore, mobile software, a popular example being mobile agents, is frequently used as a new form of building distributed network-centric applications. Tracking mobile objects, i.e., identifying their current location, is central to such settings. In this paper, we exploit the use of hierarchical distributed location databases, where each database site covers a specific geographical region and contains location information about all objects residing in it. For highly mobile objects, a scheme based on forwarding pointers enhanced with auxiliary caching techniques is presented, to reduce the cost of the overall network and database traffic generated by frequent location updates. The scheme is extended to support concurrency and failure recovery. Performance is demonstrated through a number of simulation experiments for a range of call to mobility ratios and for a variety of moving and calling behaviors.

Keywords: Mobile Computing, Distributed Database Systems, Location Management.

1 Introduction

Advances in wireless telecommunications and in the development of portable computers have provided for the emergence of wireless mobile computing [9, 10, 22]. In mobile wireless computing, hosts are not attached to a fixed geographical location, instead their point of attachment to the network changes as they relocate from one support environment to another. Besides mobility tied to wireless communications, mobile software (code or data

*University of Ioannina, Computer Science Department, Technical Report No: 99-20. A preliminary abridge version [21] of this work has appeared in CIKM98

that move among network sites) is emerging as a new form of building distributed network-centric applications. Mobile software agents [31, 1] is a popular form of such mobile software. Mobile agents are processes that may be dispatched from a source computer and transported to remote servers for execution. Mobile agents can be launched into an unstructured network and roam around to accomplish their task [2], thus providing an efficient, asynchronous method for collecting information or attaining services in rapidly evolving networks. Other applications of moving software include the relocation of a user's personal environment to support ubiquitous computing [33], and the migration of services to support load balancing, for instance the active transfer of web pages to replication servers in the proximity of clients [6].

Since delivering any message to mobile objects requires locating them first, deriving efficient strategies for tracking moving objects is central to mobile computing. In particular, for future Personal Communication Service (PCS) systems, with high user populations and numerous customer services, such signaling and database traffic for locating users is expected to increase dramatically [32]. To maintain the current location of mobile objects, distributed location databases or directories are deployed. Such databases are often hierarchically structured [32, 3, 13, 30] to accommodate the increased traffic associated with locating moving objects. In particular, each leaf database covers a specific geographical region and maintains location information for all objects currently residing in that region. Location databases at internal nodes of the hierarchy contain information about the location of all objects registered in areas covered by the databases at their children nodes.

In this paper, we consider the problem of locating moving objects using such hierarchical tree-structured location databases. In particular, to reduce the cost associated with location updates, instead of updating all location databases involved, only databases up to a specific level of the tree are updated and a forwarding pointer is appropriately set at some lower level database. However, if forwarding pointers are never deleted, then long chains are created, whose traversal results in an excessive increase in the cost of locating objects. We consider two alternative conditions for purging forwarding pointers and for updating the hierarchical database: one based on the maximum number of the forwarding pointers and the other on the physical distance between the sites in the chain. In addition, we introduce a number of auxiliary caching techniques that effectively reduce the number of pointers traversed before locating a moving object.

The scheme is extended so that it correctly supports concurrent execution of location tracking and updating. The treatment of concurrency is general enough and is applicable to any hierarchical location database. Furthermore, recovery in the case of site failures is discussed and solutions are proposed to this end.

To study the performance of the forwarding scheme in conjunction with the caching techniques and the various conditions for purging forwarding chains, we have developed an event-driven simulator. We consider two basic operations: calls to mobile objects that initiate location searches and moves of mobile objects that initiate location updates. We have performed a number of experiments for a range of call to mobility ratios and for objects with varying mobility and calling behavior. The cost of each operation has two components: a cost associated with database operations and a cost associated with messages among sites. Since which of the two components is the dominating factor follows from various system-dependent parameters, our analysis treats each of these costs separately. The results clearly show that under certain assumptions and for small call to mobility ratios, the forwarding scheme coupled with appropriate auxiliary caching techniques on a per object basis can reduce both the overall database load and the communication traffic by 60% to 20% depending on the call to mobility ratio.

The rest of this paper is organized as follows. In Section 2, we introduce the forwarding scheme and various strategies for improving its performance. In Section 3, we extend the proposed protocol to handle correctness issues that arise from the concurrent execution of location searches and updates. In Section 4, we present our models for the call and mobility behavior, briefly describe our cost model, and report performance results. In Section 5, we compare our work with related research and in Section 6, we offer conclusions.

2 The Location Strategy

To support efficient tracking of mobile objects, their current location is maintained in a distributed database. Two operations are central: updating the stored location of an object when the object moves and searching for locating the object when the object is called or invoked. The granularity of the information maintained about the location of an object varies. It can be a single site or a set of sites. In the latter case, once the set is identified, the object is located by broadcasting a message to all sites in the set; an operation called paging. We call such set of sites *logical cells* or l-cells.

In particular, in cellular architectures, mobile users are located in system-defined cells, which are bounded geographical areas. A cell is a uniquely identifiable unit. The location strategies proposed in the IS-41 and GSM [19] standards use a two-tier system in which each moving user is associated with a pair of a Home Location Register (HLR) and a Visitor Location Register (VLR). Calls to a given user, first query the VLR in the caller's region, and then if the callee is not found in this region, the callee's HLR is queried. Moves require updating the HLR. Similar approaches are readily applicable to software, for example mobile

agents. In this case, the stored location is a network site or the agent's current context. Location information may be stored at the agent's home, for example at its birth site, as well as at its visiting site, i.e., the site it is currently in. In cellular architectures, an l-cell covers a number of neighbor cells, while in other architectures an l-cell may cover few interconnected LANS.

2.1 Hierarchical Location Databases

To avoid "long-distance" signalling messages to the HLR, when most location searches and updates are geographically localized, a hierarchical directory structure has been proposed [32, 3, 13] and is under study for the European third-generation mobile system called the Universal Mobile Telecommunication System (UMTS) [8]. Similar hierarchical architectures have been proposed for locating moving objects in wireline distributed computing as well (for example, [30]).

Hierarchical location schemes extend two-tier schemes by maintaining a hierarchy of location databases, where a location database at a higher level contains location information for objects located at levels below it. A location database at a leaf serves a single l-cell and contains entries for all objects currently in in this l-cell. A database at an internal node maintains information about objects residing in the set of l-cells in its subtree. For each mobile object, this information is a pointer to an entry at a lower level database. For example, in Figure 1, for an object x residing at l-cell 18, there is an entry in the database at node 0 pointing to the entry for x in the database at node 2. The entry for x in the database at node 2 points to the entry for x in the database at node 6, which in turns points to the entry for x in the database at node 18.

It is often the case that the only way that two nodes can communicate with each other is through the hierarchy; no other physical connection exists among them. For instance, in telephony, the databases may be placed at the telephone switches.

Such hierarchical schemes lead to reductions in communication cost when most calls and moves are localized. In such cases, instead of contacting the HLR of the object that may be located far away from the object's current location, a small number of location databases in the object's neighborhood are accessed. In addition, there is no need for binding an object to a permanent home location register (HLR), since the object can be located by querying the databases in the hierarchy. On the other hand, the number of database operations caused by call and move operations increase. Table 1 summarizes some of the pros and cons of hierarchical architectures. One problem with hierarchical architectures is that the load is not evenly distributed among all nodes since more load is imposed on notes on higher levels. Approaches to distributed the entries of higher-level nodes at more than one site are

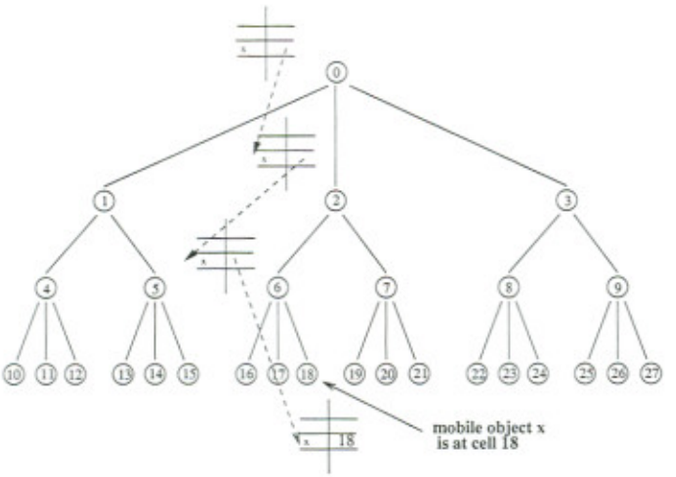

Figure 1: Hierarchical Location Schema. Location databases' entries are pointers to lower level databases.

(+)	No need for life-long numbering (no pre-assigned HLR)
(+)	Support for locality
(-)	Increased number of operations (database operations and communication messages)
(-)	Increased load and storage requirements at higher-levels

Table 1: Summary of the Pros and Cons of Hierarchical Architectures

applicable but are not discussed here (see [29]).

Let $LCA(i,j)$ stand for the least common ancestor of nodes i and j and $lca(i,j)$ be the level of the $LCA(i,j)$. Let the level of the leaves be level 0. When object x moves from l-cell i to l-cell j, the entries for x in the databases along the path from j to $LCA(i,j)$, and from $LCA(i,j)$ to i must be updated. For instance, when object x moves from 18 to 20, the entries at nodes 20, 7, 2, 6, and 18 are updated. Specifically, the entry for x is deleted from the databases at nodes 18 and 6, and an entry for x is added to the databases at nodes 2, 7, and 20. When a caller located at l-cell i places a call for a object y located at l-cell j, the lookup procedure queries databases starting from node i and proceeding upwards the tree until the first entry for x is encountered. This happens at node $LCA(i,j)$ (the least common ancestor of nodes i and j). Then, the lookup procedure proceeds downwards following the pointers to node j. For instance, a call placed from l-cell 21 to object x (Figure 1), queries databases at nodes 21, 7 and finds the first entry for x at node 2. Then, it follows the pointers to nodes 6 and 18. We call the above operations *basic move* and *basic call* respectively.

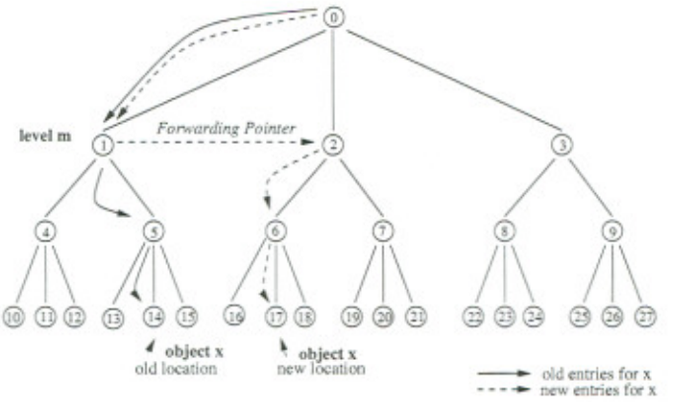

Figure 2: Pointer Forwarding.

2.2 Forwarding Pointers

To reduce the update cost, a forwarding pointer strategy is deployed. With this approach, instead of updating all databases on the path from j through $LCA(j,i)$ to i, only the databases up to a level m are updated. In addition, a pointer is set from node s to node t, where s is the ancestor of i at level m, and t is the ancestor of j at level m. The level of s and t may vary. A subsequent caller reaches x through a combination of database lookups and forwarding pointer traversals. Take, for example, object x located at node 14 that moves to node 17 (Figure 2). Let level $m = 2$. A new entry for x is created in the databases at nodes 17, 6 and 2, the entries for x in the databases at nodes 14 and 5 are deleted, and a pointer is set at x's entry in the database at node 1 pointing to the entry of x in the database at node 2. The entry for x at node 0 is not updated. Using forwarding pointers may increase the cost of calls, since it may result in a combination of tree links and forwarding pointers traversals. For instance, when an object, say at l-cell 22, calls x, the search message traverses the tree from node 12 up to the root node 0 where the first entry for x is found, then goes down to 1, follows the forwarding pointer to 2, and traverses downwards the path from 2 to 17. On the other hand, a call placed by an object at 15, results in a shorter route: it goes up to 5 and 1 and then to 2, 6, and 17.

In the following, we assume forwarding at the leaf level in which case forwarding pointers are set among leaf nodes, but the analysis applies to other types of forwarding as well. In this case, when an object moves from location i to location j, a forwarding pointer is simply set at node i pointing to node j. Calls originating from a l-cell k to an object x located at l-cell l proceed initially as in the basic call: go up the tree till the first entry for x is encountered and then down following the entries to a leaf node, say m. However, x may not be actually located at m, instead the entree at m may be a forwarding pointer to another

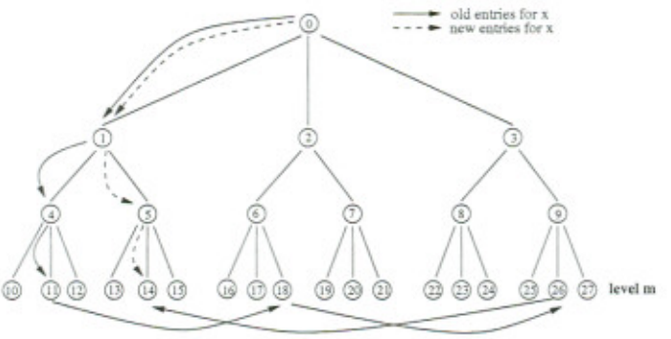

Figure 3: Full Update of the Directory Entries.

leaf node. Thus, before actually locating an object, a chain of forwarding pointers may need to be traversed.

2.3 Caching

We propose auxiliary techniques that reduce the number of forwarding pointers that a call traverses before locating a mobile object. The first technique is based on the observation that it should be possible to re-use the information about the object's location from the previous call to that object. To this end, each time a call is set up, after the address of the callee has been resolved, it is saved at the first node of the chain. Thus, any subsequent call to that object can use the cached location to locate the object, instead of traversing a number of forwarding pointers. This is essentially a variation of caching in which instead of caching the address of the callee at the caller [15], the address is cached at the starting node of the chain of forwarding pointers. The cached location can then be used not only by subsequent calls from the same l-cell, but from calls originated from any l-cell. Caching does not add to the latency of a call operation, since it can be performed off-line. In contrast to caching at the caller's location, in which case cache entries become obsolete once the callee moves, the cached entry at the first node of the chain can still be used in conjunction with the forwarding pointers even when the callee moves.

A similar caching strategy can be deployed along with move operations. In particular, during each move, the new location of the object can be cached at the first node of the chain. However, in contrast to caching at calls, the entry cached at a move is used only if the next operation involving the object is a call operation. Intuitively, if the call-to-mobility ratio (CMR) is very low, continuously updating the cached entry at each and every move operation, (which corresponds to having in effect a chain of one pointer) adds to the overall traffic without necessarily reducing the latency of some call operation. Thus, instead of caching at each move, caching is done selectively.

The condition for caching at a move is based on an estimation of the probability that the next operation will be a call operation. This is approximated by comparing the number of consequent moves and the call-to-mobility ratio. This requires a counter be associated with each mobile unit. The counter is incremented when the unit changes location and is set to zero when the unit receives a call. It also assumes that an estimation of the CMR is available. For this purpose, techniques such as those proposed in [15] can be utilized. Let $number\_of\_moves$ be the value of the counter and $estimated\_cmr$ be the value of the current CMR, then a simple heuristic is to cache when $number\_of\_moves \geq estimated\_cmr$. Again, caching can be performed off-line without adding up to the move set-up time.

2.4 Full Updates

Besides caching, in order for the hierarchical directory scheme to maintain the property of efficiently supporting local operations, the directory entries must be occasionally fully updated. Otherwise, the first node of the chain, i.e., the node at which each call is initially directed by following the tree database entries, may end up being far away from the object's actual location. Fully updating the data structure includes (a) deleting all intermediate forwarding pointers in the chain and (b) updating the internal nodes of the tree (Figure 3).

Regarding full updates, two issues must be addressed. One issue is the correct execution of calls that proceed concurrently with a full update. We consider this issue in Section 3.1. The other issue refers to the condition for initiating these procedures. We consider two possibilities. The first is to initiate a full update when the number of forwarding pointers exceeds some maximum value. This is reasonable since the size of the chain directly affects the cost of calls since the cost depends on the number of pointers to be traversed. The other is to fully update when the object moves outside the vicinity of its current region, i.e., when locality deteriorates. We capture this by considering the level of the least common ancestor of the previous and the new location. In particular, an update is initiated when the object moves to a location u such that the level of the least common ancestor of u and the previous location is larger than a threshold level. We experimented with different values for both criteria for updating. The results are presented in Section 5.3.

Before setting a forwarding pointer, detecting cycles is necessary to avoid infinite loops during calls. For example, consider chain $11 \rightarrow 18 \rightarrow 26 \rightarrow 14$ and a move made to 18. Carelessly adding a pointer to 18 results in chain $11 \rightarrow 18 \rightarrow 26 \rightarrow 14 \rightarrow 18$. Cycles can be detected by checking upon each move of an object A from j to i, whether an entry for A already exists at i. If so, there is a path from j to i which can then be purged. Thus, the chain of the example becomes $11 \rightarrow 18$.

Finally, although during normal moves, cache entries never become invalid, moves that

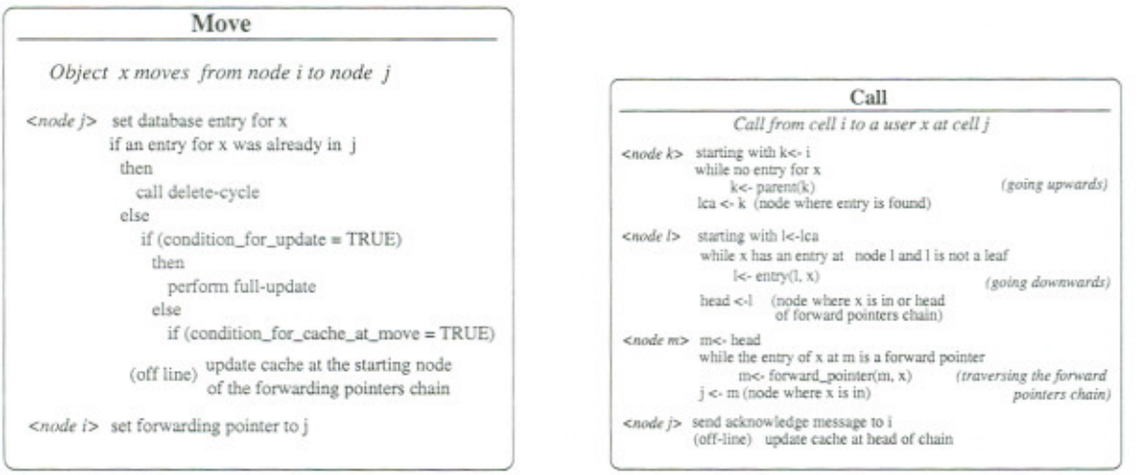

Figure 4: Move and Call Procedures. For clarity of presentation, for the call operation we do not show the case in which node i is one of the nodes in the forwarding pointers chain.

form cycles or cause the initiation of full updates make cached entries obsolete. Thus, there is an additional overhead associated with such moves, that of updating the entry cached at the first node of the chain, if any.

Figure 4 summarizes the call and move procedures. The deployment of these techniques along with the forwarding pointers strategy reduces the total database load and communication traffic over the basic hierarchical schema by a considerable factor (see Section 5).

3 Concurrency Control and Recovery

3.1 Correctness of Concurrent Operation

In any hierarchical location database, each call results in a number of query operations being issued at various nodes of the hierarchical database. Similarly, a full update initiated by a move causes update operations to be executed at several nodes of the database. The discussion so far was based on the assumption that moves and calls arrive sequentially and are handled atomically and isolated one at a time, and thus, there is no interleaving between the database queries and updates initiated by call and move operations. In this section, we extend the move and call procedures to handle concurrency.

First, when a move from l-cell i to l-cell j occurs, a forwarding pointer pointing to j is set at i to prevent from being lost any calls that have been issued prior to the movement and have been directed to the old address i. Then, *a specific order* is imposed on the database operations caused by a full update. In particular, each full update is performed in two phases: an *add* and a *delete* phase. Let an object's x movement from i to j that causes a

full update, and let v_0 be the first node in the forwarding pointers chain of x. If there is no such chain, v_0 is i. First, entries at the path from j to $LCA(j, v_0)$ are added in a bottom-up fashion and then the entries at the path from $LCA(j, v_0)$ to v_0 are deleted in a top-down fashion. In a similar manner, forwarding pointers from v_0 to i, if any, are deleted starting from v_0 and moving towards i. We also distinguish between the two phases of a call. During the *upward phase*, a call moves up the tree to find an entry for the callee. Once the first entry for the location of the callee is found, the call moves *downwards* to this location.

We serialize full updates caused by move operations as follows: a full update operations P_k caused by a movement of an object x is initiated only after any database updates caused by any previous full update operation P_m of the location of x has been completed. To enforce the serialization of full updates, P_k locks for updates the database entry for x at the new address j. When an entry is locked for updates by an operation P_k, it cannot be updated by any other operation P_l ($P_l \neq P_k$). Instead, any such update operation is blocked until the lock is released. A message is sent by P_k to unlock j, when the last entry for x is deleted. Only then, can a new full update operation P_l proceed.

Besides updates, we must also ensure that any call concurrent with a full update will succeed in locating the object. Specifically, we will first show that:

Lemma 1 *During the upward phase, any call placed by an object at l-cell c and being concurrently executed with a move, say from location i to location j, will be directed either to the old location, i, or the new location j.*

Proof. To distinguish among the possible relative positions of c, i, and j, we use the following two properties of the least common ancestor that hold for any tree nodes x, y, and z: (a) $lca(x, y) = lca(x, z) \Rightarrow lca(y, z) \leq lca(x, z)$, and (b) $lca(x, y) > lca(x, z) \Rightarrow lca(y, z) = lca(x, y)$. Case(i): $lca(j, i) < lca(i, c)$, that is the caller is equidistant from the old, i, and the new, j, locations. Then, during its upward phase, the call encounters the first entry for x at $LCA(j, c)$ ($= LCA(i, c)$). Then, at node $LCA(j, i)$, the call moves towards j, if the add phase has reached the node, and towards i, otherwise. Case (ii): $lca(j, i) > lca(j, c)$, that is the caller is closer to the new location. The call proceeds upwards to node $LCA(j, c)$. If the add phase has reached $LCA(j, c)$, then the call is directed to j. Otherwise, it continues moving upwards till it either reaches a newly added entry or it reaches $LCA(c, i)$, whichever comes first. In the former case, the caller is directed to j and in the later case to i. Case(iii): $lca(j, i) = lca(j, c)$, that is the caller is closer to the old location. If the delete phase has reached node $LCA(i, c)$, the call moves up to $LCA(j, c)$ and then towards j. Otherwise, it moves towards i. □

Now, we must show that the call during its downward phase will succeed in locating

the callee. There is a racing issue. To illustrate it, assume the concurrent execution of a full update for object x being in its delete phase and a concurrent call to object x being in its downward phase. Say that at some time, the call queries node m at level k and follows the pointer to the corresponding level $k-1$ node, say node l. Immediately afterwards, the move deletes the entry at level k. Then, traveling faster than the call, the move arrives at node l and deletes the entry for x. When the call arrives at node l, it founds no entry for x. There are at least three ways to treat the problem. We adopt the third one as being the most practical.

Method 1: Repeatable Calls

In this approach, no special treatment is given to concurrent operations. Calls that read obsolete data fail to track the object, and the lookup procedure is reissued anew. Specifically, a search fails at some node u at level k, if the callee has moved out of the subtree rooted at u and the corresponding entries have been deleted. In this case, the call can continue by moving upwards to level $k+1$ and read the database at the parent of node u. Unless, in the meanwhile, the callee has moved again in the subtree rooted at u, the call will succeed this second time. Otherwise, it will move repeatedly up and down the tree. Thus, although simple, this method does not provide any upper bound on the number of tries a call has to make before locating a moving object.

Method 2: Obtaining Read Locks

In the *transactional* approach, traditional database concurrency control techniques are used to enforce that each call and move operation is executed as a transaction, i.e., an isolated unit. This approach is highly impractical since for instance acquiring locks at all distributed databases involved in a call or move operation causes prohibitive delays.

Method 3: Using Counters

We associate with each entry for x in each location database at level k a counter, called pending calls counter (PCC_k). PCC_k counts the number of calls that queried the database entry at level k while moving downwards to locate x or following the chain of forwarding pointers towards x's current location. An additional counter at each node i, called $local_i$, counts the number of calls whose downward phase started at node i, i.e., calls that found the first entry for the callee at node i. Note, that $PCC_{k+1} + local_{k+1} - PCC_k$ is equal to the number of call operations traveling from level $k+1$ to level k.

During the delete phase of a full update operation, an entry at a level k database is

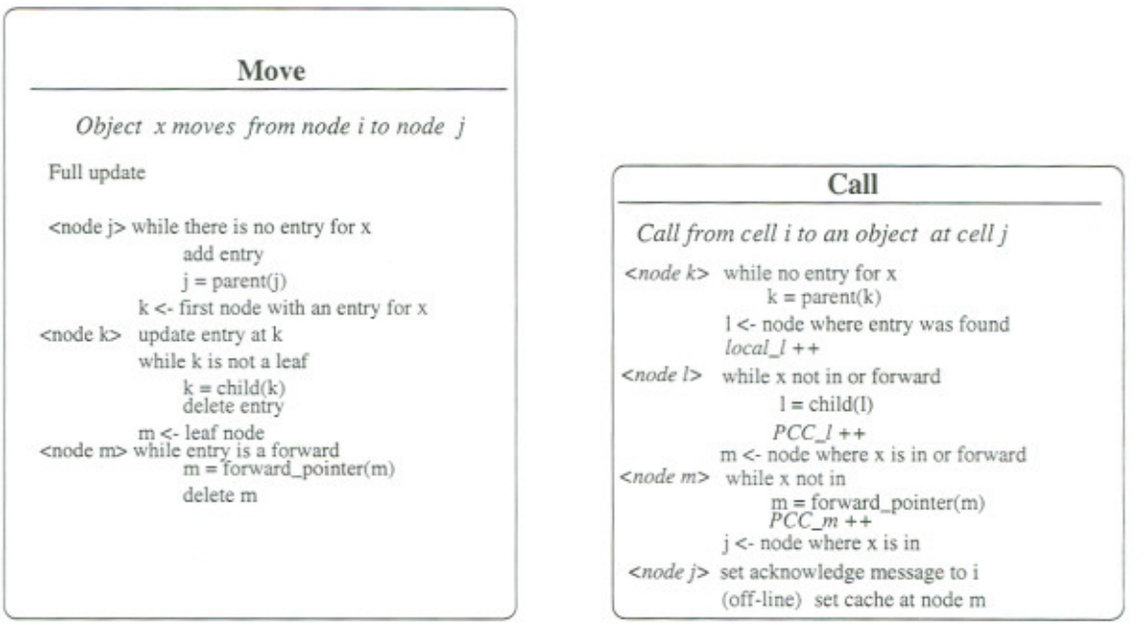

Figure 5: Move and call procedures extended for handling concurrency.

deleted only if $PCC_k \geq PCC_{k+1} + local_{k+1}$. Otherwise, the delete operation is delayed till the calls searching for x arrive at the level $k+1$ database.

Lemma 2 *During the downward phase, a call moves correctly to the location found in the upward phase. Thus, it arrives at j either directly or indirectly through i by following pointers.*

Proof. Let a call c be at level k during its downward phase after having found an entry for x. If the entry for x directs the call to the new location j, then the call will succeed in finding an entry at level $k-1$ since in the add phase the entries are included in a bottom-up fashion. If the entry directs the call to the old location i, the call will succeed finding an entry at level $k-1$ since an entry is deleted only when all pending calls from previous levels have been serviced as indicated by the PCC. Similar claims hold for following the forward chain, if any. □

Thus, all calls will eventually succeed in reaching either i or j. In effect, the algorithm in the first case makes a call appear as if it has occurred before the move, that is, it pushes the call backward, or makes it appear as if it has occurred after the move, that is, it pushes the call forward. Figure 5 summarizes the required modifications of the call procedure and the implementation of the full-update operation.

3.2 Failure Recovery

With regards to communication failures, we assume that low level communication protocols ensure that (a) there are no errors in messages and (b) no messages are lost. Next, we consider site and media failures. In the case of site failures, one or more database sites become unavailable, however, location data stored in disk are not lost and become available upon recovery. In the case of media failures, location data are permanently lost. In both cases, the replication inherent in the hierarchical model is exploited for efficient recovery.

Internal Node Failure

When an internal database site becomes unavailable, any location query for an object x directed to this site is re-directed to all children of the site. The call procedure proceeds downwards from the child at which an entry for x is found. When a database site is unreachable, any requests to insert or delete entries at this site are logged at the sending site j. The updating process is suspended till the site recovers. Following recovery, the update process resumes from site j.

Upon recovery, information at an internal node can be reconstructed by combining information from its children. To this end, node i sends a request to *reconstruct* to its children. Their replies are unioned to reconstruct the location entries at i. There is one subtle point. Consider the following scenario. A delete message for object x is sent from node i to its child j, then i fails. Shortly after, i recovers and sends a reconstruct message to j that arrives before the previously sent delete message. In this case, the reply from j and thus the reconstructed location database at i includes an entry for x. Later, when the delete message arrives at j, the entry at j will be deleted. Thus, queries for x arriving to j from i will fail to find an entry for x. A simple solution is to augment delete and reconstruct messages with timestamps. Each site j maintains the timestamp t_{rec} of the most recent reconstruct message received. If a delete message with timestamp smaller than t_{rec} arrives at j, j first sends a delete message to its parent. It deletes the entry only after the message is acknowledged.

Thus, there is no need to maintain backups of internal database sites, since even in the case of media failures, information from children database sites suffices to reconstruct the location database at the site.

Leaf Node Failure

In the case of a search arriving at a leaf database site that failed, first all sites in the l-cell covered by the failed site are paged. If the object is not found, then in the case of a search

originated from a caller at this site, the search operation proceeds upwards to the parent of the site as in the normal case. If, however, the search arrives from a parent or sibling site, indicating that the search followed a tree or forward link respectively, a global search is necessary to locate the moving object, when paging fails. Some form of informed search, such as searching sites in the neighborhood, is due. Alternatively, the call may be suspended till the database site is reconstructed. When an object enters a new l-cell or leaves an l-cell, whose covering location database has failed, the object is informed to periodically poll the failed site to install the location updates. Upon recovery after a site failures, obsolete entries created by objects entering or leaving l-cells during the failure, are expected to be updated through such periodic polls.

In the case of media failures, location databases at the leaf level must be reconstructed from scratch. Upon recovery, the database site can page all l-cells it covers to reconstruct its entries. Alternatively, entries may be reconstructed on demand, when a lookup arrives at the site. In addition, entries are added for objects in the l-cell as these objects place calls. Keeping backups can expedite the recovering procedure. However, forwarding pointers cannot be recovered as described. Instead, global searches are necessary to locate objects linked with forward pointers.

4 Other Issues

4.1 Replication

Many location strategies (e.g., citeShivakumar95,Rajagopalan95,Jannink96) involve replicating the location of a moving object in more than one site. Replication results in faster searches, since it increases the probability of finding a location entry at a neighbor site. However, faster searches are achieved with the expense of an increase in the cost of updates. When an object moves, all replicas maintaining its location become obsolete. Maintaining forwarding pointers reduces the overhead of updating replicas, since an object can still be located from its previous location following forwarding pointers. Selectively updating some of the replicas is also possible. Similarly, in the case of caching the callee's location at the caller's site (e.g., [15, 13]), if forwarding pointers are used, cache entries remain valid even when the callee moves. Experiments (see Experiment 4 in Section 5) show that when replicas are kept at selective sites, the forwarding scheme surplus the basic scheme even for high call to mobility ratios.

4.2 Advanced Location Queries

Besides location searches and updates, hierarchical location databases can be exploited for answering a wide range of advanced queries involving the location of moving objects. Such queries may be posed by mobile objects or their answer may involve mobile objects. An example is geographical multicasting [20] that involves sending messages to all objects currently in a specific geographical region. Other queries include queries for locating near-by objects (for instance mobile objects looking for specific services in their vicinity) or containment queries (for example, locating all objects in a specific region). Such queries can be efficiently answered using hierarchical databases, e..g, be querying databases at the region specified.

The introduction of forwarding pointers introduces a level of uncertainty in answering queries because location entries are not fully updated upon each move. Thus, answers to location queries are approximate. However, the uncertainty is bounded (or in the terms of [11] ignorance is bounded) depending on the condition for full updates. In particular, consider the first condition, according to which, location entries are fully updated to reflect the current location of an object, when the length of the chain of forwarding pointers exceeds some pre-specified limit k. Then the answer to a location query is guarantee to be correct in the sense that: the location returned as an answer was one of the k previous location of the object. This can be seen as a time bound, the object was in the region in one of its k previous moves. The second condition of full updates depends on the distance traveled; location entries are fully updated when the lca of the current and previous location exceeds a specified threshold x. In this case the answer to the query is within a specified spatial limit in the sense that: the object is currently in a site with a maximum distance depending on x from the location returned as an answer.

5 Performance Evaluation

We assume a hierarchy of location databases appropriately placed at the nodes of a given communication network. To allow for maximum flexibility in the design of the location management scheme, we consider hierarchies with a variable number of levels. The region covered by each leaf database corresponds to a unique physical address.

5.1 Cost Estimation

We make a distinction between the database and the communication cost. We consider as database cost the total number of database operations (queries or updates). For the

communication cost, we count the total number of links traversed by the messages involved. Assuming that two nodes u and v communicate with each other by traversing the spanning tree connecting them, let $span(u, v)$ be the number of links in the path connecting them in this spanning tree. We assume that $span(u, v) = r\, 2\, lca(u, v)$, that is, it costs r times less, (i.e., there are r less links in the path) for nodes u and v to communicate directly than to communicate through their least common ancestor in the tree. Parameter r allows the modeling of different communication infrastructures, for example when the only way for two nodes to communicate is through the links of the tree, then $r = 1$. Let c be the l-cell of caller x, v be the current l-cell of object y, k be the length of the chain of forwarding pointers for object y, and n be the new l-cell inside which y moves. In the forwarding scheme, let v_0 be the l-cell containing the first forwarding pointer and v_i $(i = 0, .., k)$ the i-th node in the chain. We consider first a call placed by object x to object y. The corresponding costs are as follows:

- *Basic Scheme:*

  database cost: $2\, lca(c, v) + 1$

  communication cost: $2\, lca(c, v) + span(c, v)$

- *Forwarding Scheme:*

  database cost: $2\, lca(c, v_0) + k + 1$

  communication cost: $2\, lca(c, v_0) + \sum_0^{k-1} span(v_i, v_{i+1}) + span(c, v)$

For a move of object y from the current l-cell v to a new l-cell n, the costs are:

- *Basic Scheme:*

  database cost $2\, lca(v, n) + 1$

  communication cost $2\, lca(v, n) + 1 + span(v, n)$

- *Forwarding Scheme:*

  database cost: 2

  communication cost: $span(v, n)$

There are additional costs for caching at calls or moves, fully updating the hierarchical scheme, and deleting cycles, as follows:

- cost of caching at calls or moves:

  communication cost: $span(n, v_0)$

  database cost: 1

- cost of a full update of the hierarchical scheme resulting from a move outside the vicinity of the current region:

  communication cost: $2\,lca(n, v_0) + \sum_0^{k-2} span(v_i, v_{i+1})$

  database cost: $2\,lca(n, v_0) + k$

- cost of deleting a cycle (where l is the length of the cycle):

  communication cost: $\sum_{i=1}^{i=l} span(v_j, v_{j+1}) + span(n, v_0)$

  database cost: l

5.2 Calling and Mobility Model

We assume that, for each object, calls and moves occur independently. The interarrival times between two calls follow an exponential distribution, with parameter the mean interarrival time between two calls, t_c. The interarrival times between two moves follow another exponential distribution, with parameter the mean interarrival time between two moves, t_m. The ratio of the number of calls over the number of moves called *Call to Mobility Ratio* (CMR) is then

$$CMR = \frac{t_m}{t_c}.$$

The source of a call event is selected using one of the following distributions:

- **Arbitrary Calls**

  A call may be placed from any l-cell with equal probability $\frac{1}{n}$, where n is the number of different l-cells. We use a discrete uniform distribution to select one from n l-cells.

- **Set of Frequent Callers**

  Each object receives most of its calls from a specific set of locations. This corresponds to a real-life situation in which an object is frequently called by a set of other objects or groups of objects, (e.g., in case of mobile users, friends, family, business associates or regular customers, or in case of mobile software, associated code). We model a set of frequent callers with a discrete bimodal distribution, which distributes a P_f probability uniformly over a set of specific locations and a $1-P_f$ probability uniformly over all other locations. So, a call has $\frac{P_f}{n_f}$ probability to be placed by a specific frequent calling location, and $\frac{1-P_f}{n-n_f}$ to be placed by another location, where n_f is the number of frequent calling locations.

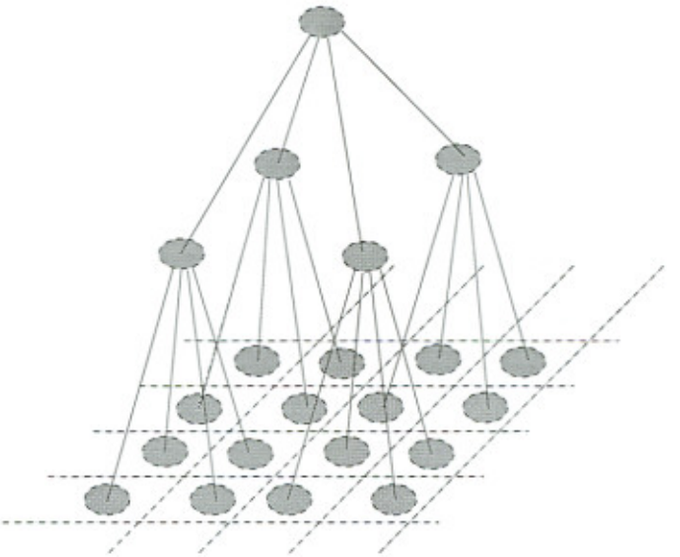

Figure 6: Movement to adjacent cells on a grid.

The destination of a move event is selected via one of the following distributions:

- **Arbitrary Moves**

  An object may move to any location, except from its current one, with the same probability. We use a uniform distribution as in the case of arbitrary calls, however the probability that the object remains in the same location after a move is 0.

- **Frequent Moves to Neighbor Cells on a Grid**

  Since objects usually move to nearby locations, we model such a situation in which distant moves are unlikely to happen and short moves to neighbor locations are most likely to happen. We assume that the object moves on a grid of l-cells (see Figure 6) and that the object may move with a high probability (P_N) to a neighbor l-cell on the grid while there is a small probability ($1 - P_N$) that the object will jump to some other non-neighbor l-cell. In case of wireless mobile computing, the movement to neighbor l-cells corresponds to an active user physically moving across l-cells. On the other hand, the rare movement to non-neighbor l-cells, corresponds to a user turning off its mobile host and turning it on again in some arbitrary l-cell. To model this distribution, we use a discrete bimodal probability distribution similar to that used for the set of frequent callers.

5.3 Experiments and Results

We have developed an event-driven simulator to evaluate the performance of the location strategies. We simulate calls to a specific mobile object and moves made by this object using an event-driven simulator. An event is either a move or a call event. The simulator software has been developed in C++ and runs on a SUN 10 workstation. The purpose of the set of experiments is to:

- determine the optimal condition for initiating full updates,
- demonstrate the benefits of caching,
- provide a comparison of the forwarding schema with the basic schema,
- illustrate optimizations of the forwarding schema for frequent callers.

We run the experiments for a wide range of call to mobility ratios and for a total of about 6000 move and call events per object. We have experimented with hierarchies of different height and width. The results show that the relative performance of the schemes under consideration remains the same. The results presented are for a tree of height 8 and out-degree 4 [28]. The experiments were performed multiple times and statistical means were derived for all estimates. Since the relative size of the communication and the database cost depends on many factors such as the load of the network and the size of the databases, we consider the two costs separately. The value of r affects the communication cost of the forwarding scheme which increases with r. For the experiments, we set the communication factor r equal to $1/3$. This is a realistic assumption, because in practice, even when traversing a forwarding pointer between two nodes means following the tree path through their LCA, it costs less, since there is no need for establishing connections and querying at each tree node on the path. Table 2 summarizes the parameters for the experiments.

Experiment 1: Condition for Full Updates

The first experiment aims at determining an appropriate condition for initiating full updates. We considered both conditions set in Section 2, i.e., (a) the condition based on the distance of the move, as determined by the level (height) of the least common ancestor of the new and old location of the move operation, and (b) the condition based on the length of the chain of forwarding pointers. In particular, we are looking for an optimal value for the level and length beyond which to fully update so that the overall database and communication cost is minimized. Table 3(a) shows the optimal such values for nearby moves, and Table 3(b)

Parameter	Description	Value
CMR	Call to Mobility Ratio for the mobile user	Varies
Database Cost	Total number of datababase (query and update) operations	Output
Communication Cost	Weighted sum of tree and forwarding pointer traversals (weighted by distance measured by the level of the lca of the two locations)	Output
nf	Number of frequent caller locations	15
Pf	Percentage of calls from frequent callers (Set of Frequent Callers Model)	85 %
Pn	Probability of moving to an adjacent location (Frequent Moves to Neighbor Cells Model)	95 %
r	Models the relative cost of following a forwarding pointer vs folowing a tree link	1/3
Topology Parameters		
Height		8
Node Outdegree		4
n	Number of leaf nodes	~ 65000

Table 2: Parameters of the Experiments.

CMR	Optimal Height		Optimal Length	
	database cost	comm cost	database cost	comm cost
0.01	8	8	20	20
0.05	8	8	13	12
0.1	8	7	11	8
0.2	8	7	8	5
0.3	8	7	5	4
0.4	8	6	5	3
0.5	8	6	3	2
0.6	8	4	3	2
0.8	8	4	3	2
1.0	8	4	2	1

(a)

CMR	Optimal Height		Optimal Length	
	database cost	comm cost	database cost	comm cost
0.01	8	8	20	20
0.05	8	8	20	10
0.1	8	8	14	6
0.2	8	8	13	6
0.3	8	8	9	4
0.4	8	8	8	3
0.5	8	8	7	3
0.6	8	8	6	2
0.8	8	8	5	2
1.0	8	8	5	2

(b)

Table 3: Optimal values for purging in terms of the overall database and communication cost (a) for nearby-moves and (b) for random moves.

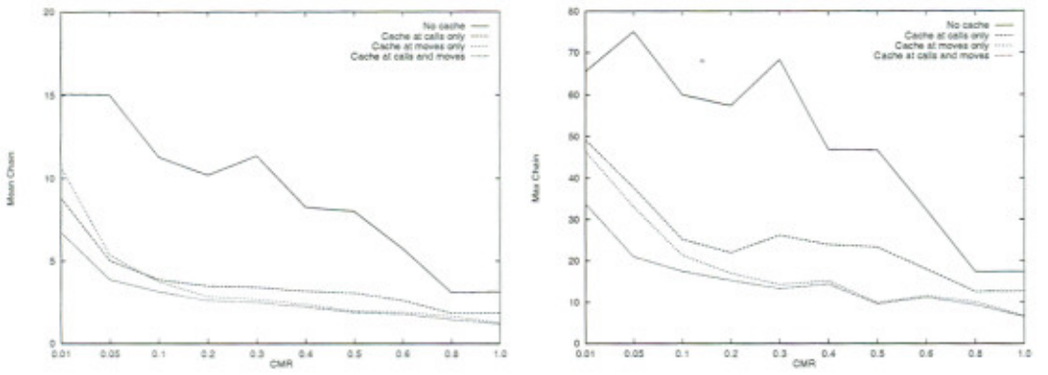

Figure 7: Effectiveness of caching for near-by moves.

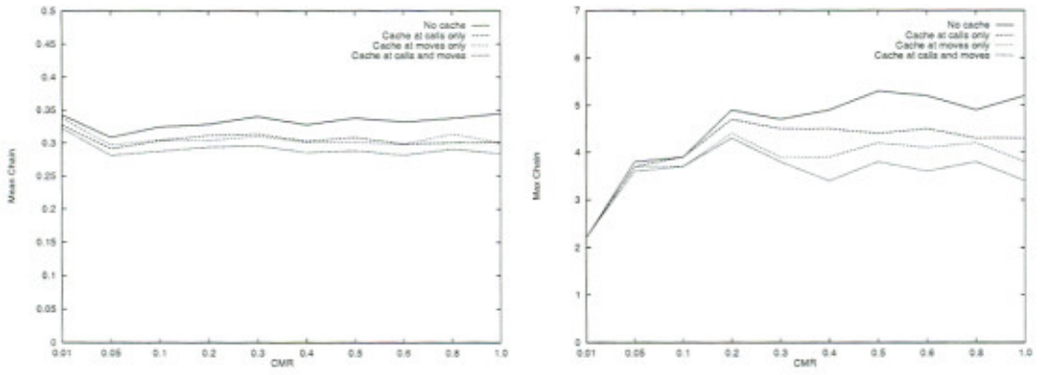

Figure 8: Effectiveness of caching for uniform moves.

for uniform moves. As expected, this value directly depends on the CMR, since frequent updates decrease the cost of calls while increasing the cost of moves. Also, the optimal values are slightly higher for uniform moves. This is because basic moves cost more in the uniform model of moves than in the near-by model of moves and thus the savings from forwarding are correspondingly higher.

Experiment 2: Benefits of Caching

We have performed a set of experiments to verify the benefits of caching. For this set of experiments, the condition for fully updating was based on the height, and the optimal values were taken from Experiment 1. Caching adds no overhead to the latency of move or call operations, since it can be performed off-line. Figure 7 demonstrates the effectiveness of caching for near-by moves and Figure 8 for uniform moves. The mean and maximum chain lengths refer to the mean and maximum number of pointers a call has to follow before locating the callee. As shown, caching proves very effective in minimizing both the mean and the maximum values of the chain. In addition, caching reduces the call set-up time

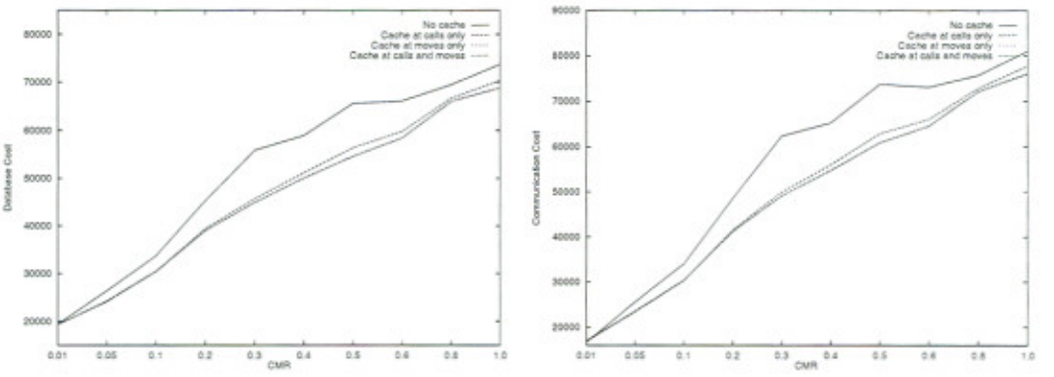

Figure 9: Overall database and communication cost for near-by moves.

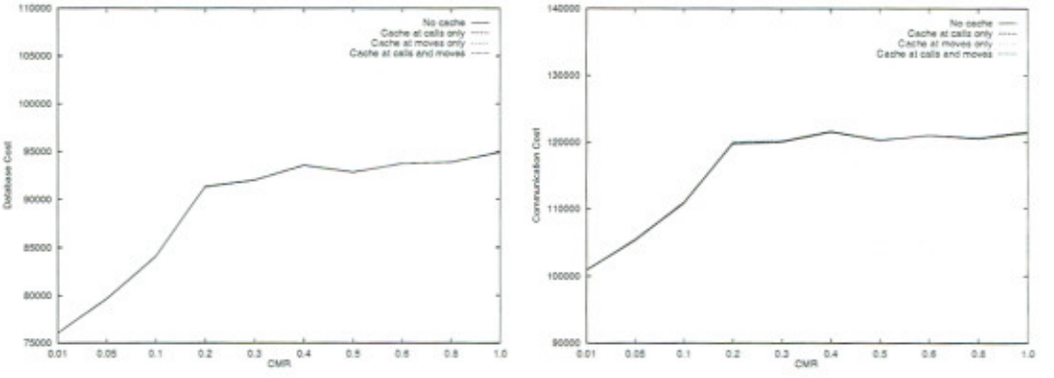

Figure 10: Overall database and communication cost for uniform moves.

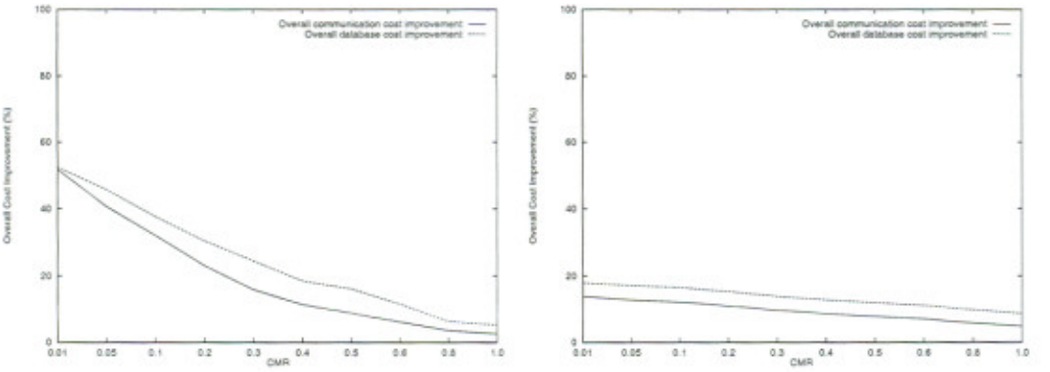

Figure 11: Decrease of the database and communication traffic caused by forwarding (left) for near-by moves and (right) for uniform moves.

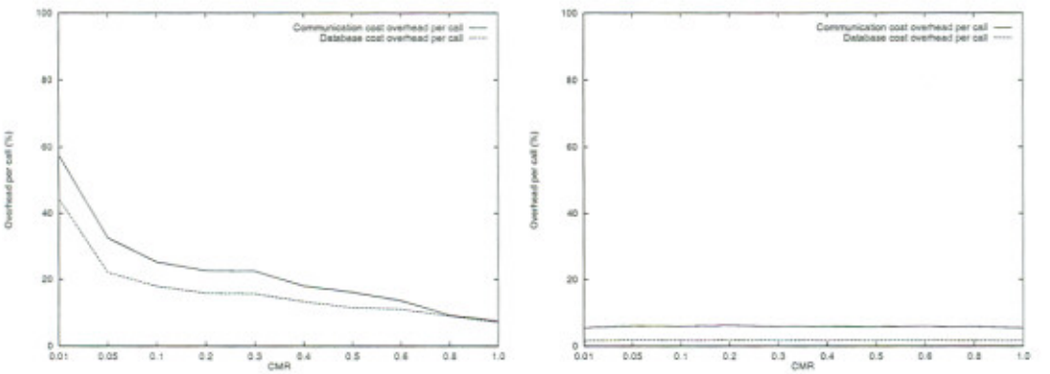

Figure 12: Increase of the call set-up time (left) for near-by moves and (right) for uniform moves.

without adding to the overall database and communication cost as demonstrated by Figure 9 for near-by moves and Figure 10 for uniform moves.

Experiment 3: Forward vs Basic

We have performed a number of experiments to study the effect of the forwarding scheme in decreasing the overall database load and communication traffic. Depending on the CMR both the overall database traffic and the communication cost are decreased by a factor of 52% to a factor of 5% for near-by moves (Figure 11(a)). For uniform calls, the corresponding values ranges from around 18% to around 7% (Figure 11(b)).

The price for this improvement is an increase of the call set up time. This is shown in Figure 12(a) for near-by moves and in Figure 12(b) for uniform moves. Call latency can be effectively kept short if information about the calling behavior is known before hand. In this case, a simple scheme that caches the location of the object at the callers can reduce

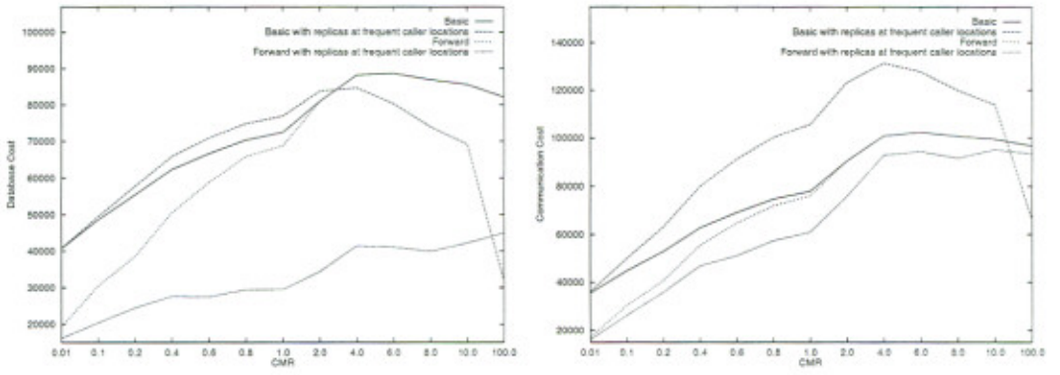

Figure 13: Overall database and communication cost for set of frequent callers.

the cost. Next, we present results for such a scheme.

Experiment 4: Set of Frequent Callers

For the purposes of this experiment, we assume a pre-specified set of frequent callers, uniformly chosen among all l-cells. In practice, either each mobile object explicitly specifies this set, or the set is derived by observing for each mobile object the frequency of receiving calls.

We have extended both the basic and the forwarding schemes with lazy replication. In particular, the location of a mobile object is replicated at the l-cell of its frequent callers. To locate an object, a frequent caller first uses the location saved at its region. If this location is outdated (e.g., the object has in between moved to a new location) a location miss is signaled. Upon a miss, the caller uses the usual procedure to locate the callee. A outdated replica of an object is updated upon a location miss, after the current location of the object is found by using the normal procedure. Setting forwarding pointers improves the performance of replication since it decreases the location miss ratio. Figure 13 shows the benefits of forwarding in terms of the overall cost, while Figure 14 shows the benefit of forwarding in terms of the call set-up time. Note, that in the case of frequent callers, the forwarding scheme works well even for high values of the CMR.

6 Related Work

Location management has attracted much current research in mobile computing. Various approaches have been proposed for reducing the cost of move and call operations for both hierarchical and non hierarchical location database architectures [22, 18]. See for example [23] for a survey.

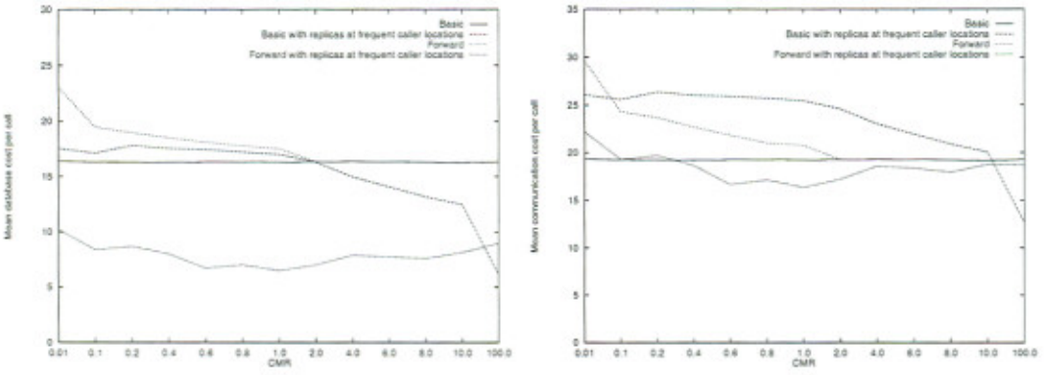

Figure 14: Call set-up time for set of frequent callers.

One proposal to reduce the cost of calls is based on replicating (see e.g., [26, 24] for hierarchical and [16] for non hierarchical architectures) or caching (see e.g., [15] for non hierarchical and [13] for hierarchical architectures) the location of mobile users at the sites of their frequent callers. To reduce the cost of network traffic generated by frequent updates, a scheme based on partitioning is presented in [5, 7]. Partitioning is performed by grouping the cells among which a mobile unit moves frequently and separating the cells between which it relocates infrequently. Only moves across partitions are reported. A hierarchical method for location binding in wide-area systems is used in the Globe wide-area location service [30, 29]. Globe advocates a combination of caching and partitioning. An interesting feature is that an object, being a service, may have more than one locations. Finally, the assignment of location databases to the nodes of a signaling network is discussed in [3], where database placement is formulated as an optimization problem. Those schemes are orthogonal to the address forwarding strategy and can be used in conduction with it to further reduce database and communication costs.

Address forwarding for a non hierarchical architecture is discussed in [14] along with an analysis of its benefits for different CMRs. Forwarding pointers have been also used in [12] again for non hierarchical location databases. The focus there is on IP-based protocols. In [4], forwarding is explored in a different framework for a hierarchical, though not tree-structured, location database architecture, called regional matching directory. The treatment is theoretical and based on a worst case order analysis.

A non-hierarchical forwarding scheme, SSP chains [25], has also been proposed for transparently migrating object references between processes in distributed computing. The SSP-chain short-cutting technique is somewhat similar to our caching at calls method.

In [17], forwarding pointers are used in hierarchical location databases. However, the architectural assumptions are different in that the actual location is saved at each internal

database instead of a pointer to the corresponding lower level database. The forwarding pointer is set at different levels in the hierarchy and not necessarily at the lower level database as in our scheme. The emphasis there is on choosing an appropriate level for setting the forwarding pointers and on updating obsolete entries in the hierarchy after a successful call.

Databases that record the location of moving objects are also discussed in [27, 34]. However, in this work, one centralized database is considered. The focus is on determining when to update the location of a moving object. The approach advances an information cost model that captures uncertainty, deviation and communication.

7 Summary

Tracking objects is central to mobile computing. In this paper, we have studied the use of forwarding pointers in a hierarchical database arrangement. In such schemes, instead of fully updating the hierarchical database upon each location update, a forwarding pointer to the new location is set at the previous location of the object. However, the forwarding scheme increases the cost of location searches, since a chain of forwarding pointers may have to be traversed before locating an object. To effectively reduce the length of such chains, we have introduced caching techniques as well as conditions for initiating an update of all associated database entries. Finally, we have described a synchronization method to control the concurrent execution of location searches and updates and a protocol to handle site failures. Experimental results show that under certain assumptions and for small call to mobility ratio, the proposed forwarding scheme leads in reduction of both the overall database load and the communication traffic by 20% to 60% depending on the call to mobility ratio and the moving behavior.

References

[1] Special Issue on Intelligent Agents. *Communications of the ACM*, 37(7), 1994.

[2] Special Issue on Internet-based Agents. *IEEE Internet Computing*, 1(4), 1997.

[3] V. Anantharam, M. L. Honig, U. Madhow, and V. K. Kei. Optimization of a Database Hierarchy for Mobility Tracking in a Personal Communications Network. *Performance Eavaluation*, 20:287–300, 1994.

[4] B. Awerbuch and D. Peleg. Concurrent Online Tracking of Mobile Users. In *Proceedings of SIGCOMM 91*, pages 221–233, November 1991.

[5] B. R. Badrinath, T. Imielinski, and A. Virmani. Locating Strategies for Personal Communications Networks. In *Proceedings of the 1992 International Conference on Networks for Personal Communications*, 1992.

[6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enchancing the Web's Infrastructure: From Caching to Replication. *IEEE Internet Computing*, 1(2):18–27, March 1997.

[7] G. Cho and L. F. Marshall. An Efficient Location and Routing Schema for Mobile Computing Environments. *IEEE Journal on Selected Areas in Communications*, 13(5), June 1995.

[8] C. Eynard, M. Lenti, A. Lombardo, O. Marengo, and S. Palazzo. A Methodology for the Performance Evaluation of Data Query Strategies in Universal Mobile Telecommunication Systems (UMTS). *IEEE Journal on Selected Areas in Communications*, 13(5), June 1995.

[9] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(6):38–47, April 1994.

[10] T. Imielinksi and B. R. Badrinath. Wireless Mobile Computing: Challenges in Data Management. *Communications of the ACM*, 37(10), October 1994.

[11] T. Imielinski and B. R. Badrinath. Querying in Highly Mobile Distributed Environmnets. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB 92)*, 1992.

[12] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr. IP-based Protocols for Mobile Internetworking. In *Proceedings of the ACM SIGCOMM Symposium on Communications, Architectures and Protocols*, pages 235–245, September 1991.

[13] R. Jain. Reducing Traffic Impacts of PCS Using Hierarchical User Location Databases. In *Proceedings of the IEEE International Conference on Communications*, 1996.

[14] R. Jain and Y-B. Ling. A Auxiliary User Location Strategy Employing Forwarding Pointers to Reduce Network Impacts of PCS. *Wireless Networks*, 1:197–210, 1995.

[15] R. Jain, Y-B. Ling, C. Lo, and S. Mohan. A Caching Startegy to Reduce Network Impacts of PCS. *IEEE Journal on Selected Areas in Communications*, 12(8):1434–44, October 1994.

[16] J. Jannink, D. Lam, N. Shivakumar, J. Widom, and D. C. Cox. Efficint and Flexible Location Management Techniques for Wireless Communication Systems. In *Proceedings of the 2nd ACM International Conference on Mobile Cmputing and Networking (Mobicom'96)*, 1996.

[17] P. Krishna, N. H. Vaidya, and D. K. Pradhan. Static and Dynamic Location Management in Mobile Wireless Networks. *Journal of Computer Communications (special issue on Mobile Computing)*, 19(4), March 1996.

[18] S. Mohan and R. Jain. Two User Location Strategies for Personal Communication Services. *IEEE Personal Communications*, 1(1):42–50, 1st Quarter 1994.

[19] M. Mouly and M. B. Pautet. The GSM System for Mobile Communications. Published by authors, 1992.

[20] J. C. Navas and T. Imielinski. Geographic Addressing and Routing. In *Proceedings of the 3rd ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'97)*, pages 26–30, Budapest, Hungary, September 1997.

[21] E. Pitoura and I. Fudos. An Efficient Hierarchical Scheme for Locating Highly Mobile Users. In *Proceedings of the 7th Iternational Conference on Information and Knoweledge Management (CIKM'98)*, pages 218–225, November 1998.

[22] E. Pitoura and G. Samaras. *Data Management for Mobile Computing.* Kluwer Academic Publishers, 1998.

[23] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. Technical Report TR: 98-20, Univ. of Ioannina, Computer Science Dept, 1998. IEEE TKDE, Accepted. Also available at:www.cs.uoi.gr/~ pitoura/pub.html.

[24] S. Rajagopalan and B. R. Badrinath. An Adaptive Location Management Strategy for Mobile IP. In *Proceedings of the 1st ACM International Conference on Mobile Cmputing and Networking (Mobicom'95)*, Berkeley, CA, October 1995.

[25] M. Shapiro, P. Dickman, and D. Plainfosse. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report Technical Report 1799, INRIA, Rocquentcourt, France, November 1992.

[26] N. Shivakumar and J. Widom. User Profile Replication for Faster Location Lookup in Mobile Environments. In *Proceedings of the 1st ACM International Conference on Mobile Cmputing and Networking (Mobicom'95)*, 161-169, October 1995.

[27] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the 13th International Conference on Data Engineering (ICDE 97)*, 1997.

[28] Stanford Pleiades Research Group. Stanford University Mobile Activity TRAces (SUMATRA). www-db.stanford.edu/sumatra.

[29] M. van Steen, F. J. Hauck, G. Ballintijin, and A. S. Tanenbaum. Algorithmic Design of the Globe Wide-Area Location Service. *The Computer Journal*, 41(5):297–310, 1998.

[30] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 2–7, January 1998.

[31] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*. Springer Verlag, LNCS 1222, 1997.

[32] J. Z. Wang. A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems. *IEEE Journal on Selected Areas in Communications*, 11(6):850–860, August 1993.

[33] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[34] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proceedings of the 14th International Conference on Data Engineering (ICDE 98)*, 1998.