Architecture of a Large-scale Location Service

Authors:
Dipl.-Inf. A. Leonhardi
Prof. Dr. K. Rothermel

Institute of Parallel and Distributed
High-Performance Systems (IPVR)
Department of Computer Science
University of Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart
Germany

Architecture of a Large-scale Location Service

A. Leonhardi, K. Rothermel

Technical Report 2001/01
January 2001

Architecture of a Large-scale Location Service

Alexander Leonhardi and Kurt Rothermel

Institute of Parallel and Distributed
High-Performance Systems (IPVR)
University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart, Germany
alexander.leonhardi@informatik.uni-stuttgart.de

Technical Report 2001/01
Department of Computer Science
University of Stuttgart
January 2001

Abstract

Many mobile applications require some knowledge about the current geographic locations of the mobile objects involved. Therefore, services exist that can store and retrieve the position of mobile objects in an efficient and scalable way. More advanced location-aware applications, however, require additional functionality, like determining all mobile objects inside a certain geographic area (range query). This functionality is not supported by existing services on a large scale yet. In this paper, we present a generic large-scale location service. We describe the location service model, defining the semantics of position, range and nearest neighbor queries. A hierarchical distributed architecture is presented, which can efficiently process these queries, and the structure of a main-memory database for efficiently storing and retrieving position information on a location server. Finally, through measurements on a first prototype of this architecture, we show the feasibility of such a location service.

1 Introduction

Location-aware services provide the base for a wide range of promising application areas, such as navigation, sentient computing (e.g., [18]), and situated information spaces (e.g., [2]). In particular, the latter kind of application is already emerging into Personal Communication Services (PCS), where users may retrieve (situated) information that is “close” to their current location.

In order to support a wide rage of location-aware applications, a generic locations service is needed, which maintains the locations of tracked objects, like persons or vehicles. While for some applications it might be sufficient to retrieve the current position of a given object, others might require more sophisticated types of queries like determining all mobile objects that are inside a certain area (range queries) or the mobile object nearest to a certain location (nearest neighbor query). For example, in a city guide application an information service for public transportation might want to announce the delay of a bus to all users waiting at the next station. In consequence, a user may want to find the nearest available taxi cab. Besides a query/result-style of communication the location service should also support event-based interaction. In other words, applications should be able to register for predicates, such as “more than five objects are in a certain area” or “two users of the system meet”, at the location service, which asynchronously informs the registered applications when the predicate becomes true (see [2]).

Position sensing devices, such as GPS differ in the accuracy with which they record information. While GPS is accurate to within 10 m, an indoor location system might have a finer resolution, for example, the Active Bat system [8] offers an accuracy of up to 10 cm. Consequently, the position information maintained by a location service may differ in accuracy if it integrates various sensor systems. When designing an API for the location service this fact must be considered to allow clients and tracked objects to specify the requested accuracy. Of course, query processing algorithms must take into account the varying accuracy of the recorded position information as well as the accuracy related QoS requested for the query. Moreover, different positioning systems can deliver the same information but in different forms. For example, an Active Badge System [23] delivers position by means of cell identities, while GPS is based on a geographic coordinate system. A location service implementation should hide this heterogeneity as far as possible from applications.

The privacy of the tracked object's recorded position information will be crucial for the acceptance of such a service. Therefore, authentication and authorization mechanisms must be integrated into a location service (for detailed discussion of this aspect see [11]). In addition, we think that it is important to allow tracked objects to control the accuracy of the position information that is submitted to the location service ("I am in town" vs. "I am at the central station"). In other words, independent of the underlying sensor system users should be able to specify bounds on the accuracy information transferred to the location service. Moreover, a user should be able to change these bounds on the fly depending on his or her current situation. Consequently, the accuracy of the recorded position information not only depends on the underlying sensor systems but also on the users' privacy needs and trust in such a service.

We believe that the location-awareness of applications will not be limited to indoor or outdoor situations but will become a global issue, which is supported by a location service integrating various indoor and outdoor positioning systems. We further believe that many applications will become location-aware as soon as the required infrastructure gets available. Consequently, we expect that a global location service must be able to handle hundred thousands of tracked objects and clients concurrently. In other words, the scalability of such a service becomes a major design goal.

In this paper, we will consider only some of the issues raised above. In particular, we will focus on architectural aspects of a large-scale location service. The main contributions of this paper are as follows: (1) A location service model (or generic API) including position, range, and nearest neighbor queries is given. The semantics of these queries are defined taking into account accuracy related QoS requirements. (2) A hierarchical server architecture for implementing the location service together with the algorithms for position updates and query processing are proposed. (3) The structure of a main-memory database for recording and accessing the position information in an efficient manner is described, and finally (4), the feasibility of the proposed architecture is shown by means of measurements based on our prototype implementation.

The remainder of the paper is organized as follows: In Section 2, we discuss related work and then present our service model in Section 3. The proposed architecture of the location service is given in Section 4, while data storage issues are covered in the subsequent section. In Section 6, the major algorithms for update and query processing are presented in detail. Section 7 shows the results of our experiments, and finally the paper concludes with a brief summary and an outlook on future work.

2 Related Work

One of the first systems providing location-awareness has been based on the Active Badge in-door locating system [23]. In this context, a location service has been developed that manages the location information for an installation of the Active Badge system [7] or its much more accurate successor, Active Bat [8]. The latter system provides an efficient event mechanism based on the notion of geometric containment. In the ParcTab system, which uses a location tracking technique similar to the Active Badge system, the location information of each user is managed and protected by a special user agent because of privacy considerations [19].

Moving Objects Databases (MODs) are databases which contain information about moving objects and their location. They are, for example, used for fleet management of transportation vehicles. Research issues are the definition of a suitable data model, querying and updating of the location information and the indexing of the data. In the DOMINO project [24], for example, the reduction of position updates by using different dead-reckoning policies is discussed. Spatio-temporal databases deal with changes of spatial information over time and additionally allow historical queries (see for example [5]).

Much research has been done on efficient, scalable location management in Personal Communications Services, where a distributed location management component is used for storing and retrieving the current communication cells for a large number of mobile phones. Currently, in GSM [14] the location information of a mobile phone is stored in the Home Location Register it is assigned to and in a Visitor Location Register responsible for its current location area. To improve scalability, hierarchical concepts and user profile replication have been proposed for future Personal Communication Services (see for example [9] or [10]). Although many concepts of location management can be used for the location service, location management is primarily concerned with finding a certain mobile phone and does not consider range or nearest neighbor queries. Also, position information is only captured with the granularity of a cell of the underlying communication network.

A location service for locating mobile objects (mainly software objects) worldwide is being developed as part of the Globe project [20]. To achieve the scalability for the desired very large number of mobile objects, the service uses a hierarchical search tree, which allows to store addresses at different levels, and caching mechanisms. However, the Globe location service returns the contact address of a queried object rather than a geographic position and does not support range or nearest neighbor queries.

Leonhardt [11] proposes a wide-scale distributed location service that is independent of application and sensor systems. In his PhD-thesis, he examines fundamental issues of such a location service and classifies them in an abstract service model. In addition, he proposes a location model for the integration of different types of sensor data and requirements as well as policies for access control of a location service. The thesis also discusses architectural aspects of a global, general-purpose location service but does not propose or evaluate a specific architecture. Maaß [12] describes the data model and access protocols for a location service based on the X.500 directory service. This service supports position, range, and nearest neighbor queries as well as notifications for predefined areas.

3 The Location Service

In our service model we distinguish between the *location service* (*LS*), which is typically implemented by a number of servers, *tracked objects*, whose position information is managed by the LS, and *clients* that query their position information through the LS. The LS is responsible for managing position information of mobile objects inside a certain *service area*. Those mobile objects whose position information is managed by the LS are denoted as *tracked objects*. The position information for the tracked objects is determined through location sensor systems and is continuously updated in the LS's database. The position of a tracked object can be determined either by a *positioning system* attached to the mobile device, such as a GPS sensor, or by an external stationary *tracking system*, like the Active Badge system (for an overview of location sensors see for example [11]). Clients exploit the LS by issuing position or range queries, where a mobile device may and often will have both roles, tracked object and client. Mobile devices are assumed to communicate with the LS by means of wireless communication, such as GSM or a Wireless LAN according to the IEEE 802.11 standard.

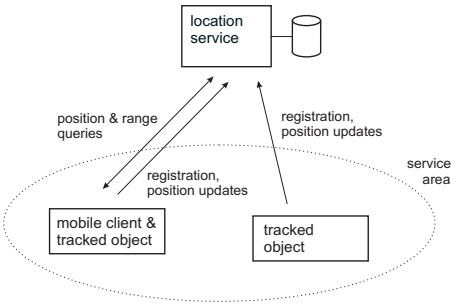

Figure 1: Basic components and interactions of the LS.

From a client's point of view, the LS maintains in its database for each tracked object o ($o \in O$) a so-called *location descriptor ld*, which includes o's position together with accuracy information. We assume position information to be based on geographic coordinate systems, such as WGS84 [16], which is used by GPS. Due to communication delays and the limited precision of sensor systems, the LS may maintain position information only up to a certain accuracy. The achievable accuracy depends on various factors, mainly on the used position update protocol, the update frequency and the underlying sensor system (for details see [15]).

A tracked object o's location descriptor, denoted as *ld(o)*, consists of two components:

ld(o).pos:	The position stored for o, given by a geographic coordinate.
ld(o).acc:	This field specifies the accuracy, which is defined to be the worst-case deviation of *ld(o).pos* from o's actual position.

Consequently, if $rp(o)$ denotes o's real position then:

$$DISTANCE(ld(o).pos - rp(o)) \leq ld(o).acc$$

As illustrated in Fig. 2, o is guaranteed to reside in the circular location area defined by *ld(o)*. Obviously, the smaller the value of *ld(o).acc* the higher is the accuracy. When an object is registered, the worst-case accuracy for this object's position information can be negotiated with the LS (see below).

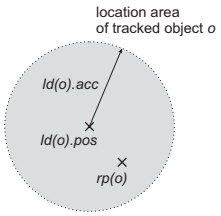

Figure 2: Location area defined by *ld(o)*.

3.1 Registration and Position Update

When an object is registered or a tracked object's position information is updated, a so-called *sighting record* is sent to the LS. A sighting record $s \in S$ has the following properties:

s.oId:	identifier of the tracked object, which is unique in the LS's namespace *OId*.
s.t:	Timestamp of the sighting.[1]

s.pos:	Object's position at time *s.t* given by a geographic coordinate. *Pos* denotes the set of possible positions.
$s.acc_{sens}$:	Sensor accuracy of the corresponding sensor system[1], which is defined to be the maximum distance between the returned position and the actual position of the tracked object at *s.t*.

To become a tracked object, a mobile device must register with the LS by issuing the $register_{LS}$ operation:

register(s, desAcc, minAcc) → *offeredAcc*

The registering instance supplies an initial sighting record *s* for the object to be registered. Moreover, it specifies the requested accuracy range by means of *desAcc* and *minAcc*, the desired and minimal accuracy, respectively. The acceptable accuracy range mainly depends on the applications considering the tracked object as well as the object's type. On the other hand, as mentioned above, the accuracy provided by the LS is limited due to the sensor infrastructure and performance reasons. It should be noted that the level of accuracy has a strong impact on the update frequency needed to keep the position up to date. Registration succeeds, if the LS can provide an accuracy value within the requested range, and fails otherwise. The offered accuracy is returned in *offeredAcc*.

After registration the LS can be requested to change the accuracy by issuing:

changeAcc(o, desAcc, minAcc) → *offeredAcc*

Whenever the currently offered accuracy changes, the LS sends a notification to the registering instance, indicating the now available accuracy.

notifyAvailAcc() → *offeredAcc*

To update the sighting records stored in the LS's database, a tracked object or stationary tracking system sends update requests according to a given update protocol (for details see [15]) to the LS.

update(s)

The LS forgets about a tracked object when *deregister* is received for this object.

deregister(o)

3.2 Position and Range Queries

The LS basically supports three types of queries, position queries, range queries and nearest neighbor queries. A position query is used to retrieve the current position of a given tracked object. In a fleet management system, for example, this type of query could be used to get the current position of a certain truck, which has been scheduled for an inspection at short notice. A range query determines all tracked objects inside a certain geographic area, $a \in A$. Such an area can be defined as an arbitrary connected polygon given by the geographic coordinates of its corners. An application in the area of fleet management would be to find all trucks that are in a given part of a city. The nearest neighbor query returns the object with the minimal distance to a given geographic position. It could be used to find the nearest (free) truck for a load of goods.

A client can retrieve the current position of an object, say *o*, by issuing:

posQuery(o) → *ld*

As response, object *o*'s location descriptor *ld(o)* is returned, with the currently offered accuracy, if *o* is a tracked object of the location service.

A range query returns the location descriptor for each tracked object located inside the indicated geographic area. Since the position information recorded for a tracked object is a circular area rather than a point it is not always clear whether or not an object belongs to a given area. In the example depicted in Fig. 3 it is clear that object o_1 is within area *a* and o_2 is outside of it. What about objects o_3 and o_4? The location areas associated with these two objects overlap with the queried area to different degrees. Clearly, the stronger the overlap the higher is the probability that the object's real position is within the area[2]. We allow the client to specify the degree of overlap that is needed to qualify for a range. Let *a* be a queried area and *ld*(*o*) specify the circular location area of object *o*, then the degree of overlap is defined to be:

$$Overlap(a, o) = SIZE(a \cap ld(o)) / SIZE(ld(o))$$

Obviously, the overlap degree can be in the range of [0,1].

---

1. For this timestamp we assume synchronized clocks, which can, for example, be achieved by using the very accurate time provided by a GPS receiver.

1. *s.t* and $s.acc_{sens}$ together with the maximum speed of the tracked object are used by the LS to obtain a bound for the accuracy at a time $t > s.t$ (see [15]).

2. We assume that the probability of an object being at a certain point is uniformly distributed over the object's location area.

Another parameter that may strongly affect the outcome of the query is the accuracy of the objects' position information. Assume, for example, that the accuracy associated with an object is 200 m, while the specified area has the size of a room. Even if the location area of the object and the range overlap to a high degree the probability that this object is within that area might be rather low. Therefore, in addition to a threshold for the overlapping degree a client should be able to request a certain degree of accuracy. Objects, whose position accuracy is below this degree, are not considered during query processing. In our example depicted in Fig. 3, object o_5 is not considered as its accuracy is below the threshold.

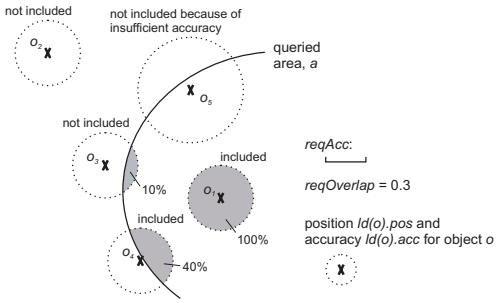

Figure 3: Example of a range query.

A range query is issued by the following operation:

rangeQuery(a, reqAcc, reqOverlap) → *objSet*

Parameter *a* identifies the geographic area, and *reqAcc* and *reqOverlap* specify the requested degree of accuracy and overlap, respectively. Clearly, *reqOverlap* must be in (0,1]. In *objSet*, the (id, location descriptor)-pairs for the objects qualifying for that query are returned:

$objSet = \{ (o, ld(o)) \mid o \in O \text{ and } Overlap(a, o) \geq reqOverlap > 0 \text{ and } ld(o).acc \leq reqAcc \}$

A nearest neighbor query selects the tracked object with the minimum distance to a given position from the LS database. It is issued by the following operation:

neighborQuery(p, reqAcc, nearQual) → *(nearestObj, nearObjSet)*

For the selected object, say *o*, the operation returns in *nearestObj* the corresponding (*o*, *ld(o)*)-pair. For *o* holds that - according to its position recorded in *ld(o).pos* - no other tracked object is closer to position *p*. As with range queries, a client may specify an accuracy threshold *reqAcc*, where objects whose accuracy value is greater (i.e., whose accuracy is worse) than this threshold are not considered for query processing:

$nearestObj = (o, ld(o)) \text{ with } o \in O \text{ and } DISTANCE(ld(o).pos, p) \leq DISTANCE(ld(o').pos, p) \; \forall \; o' \in O$ and $ld(o).acc, ld(o').acc \leq reqAcc$

Consequently, it is guaranteed, that no object fitting the requested accuracy has a distance to *p* of less than *DISTANCE(ld(o).pos, p) - reqAcc* (see Fig. 4). A client can use this lower bound for the distance to the nearest object, for example to decide on the maximum power it can use for wireless transmission without causing interference.

Note that due to the limited accuracy of the recorded position information, it cannot be guaranteed that the selected object is actually the nearest neighbor. For example, in the scenario depicted in Fig. 4 object o_1 may be nearer to *p* than *o*, if its actual position inside the service area is in the part nearer to *p* and the actual position of *o* is in the farther outside part of its service area. However, it can be shown that from all tracked objects this object is selected that is the nearest neighbor with the highest probability. (Actually the probability also depends on the radii of the location areas, but their influence is very low if the distance to p is large in comparison to them. If, for example, the distance is four times as high, their influence is less than 1 percent.[1])

In particular, due to this uncertainty some applications might be interested not only in the nearest neighbor but also in other "near" neighbors, where parameter *nearQual* qualifies what "near" means. Let *o* be the selected nearest neighbor, then *nearObjSet* is defined as follows:

$nearObjSet = \{ (o', ld(o')) \mid o' \in O, o' \neq o \text{ and } DISTANCE(ld(o').pos, p) \leq DISTANCE(ld(o).pos, p) + nearQual)$ and $ld(o').acc \leq reqAcc \}$

Consequently, if *nearQual* is set to zero, then *nearObjSet* is the empty set. On the other hand, if *nearQual* is set to $2 \times reqAcc$, it is guaranteed that all tracked objects that could potentially be closer to *p* than *o* are included in *nearObjSet*. In the example illustrated in Fig. 4, object o_1 is in *nearObjSet*, while o_2 is not, as the position of o_2 lies outside of the circle around *p* given by adding *nearQual* to the distance of the selected object *o* from *p*. Object o_3 is again not considered because of its accuracy.

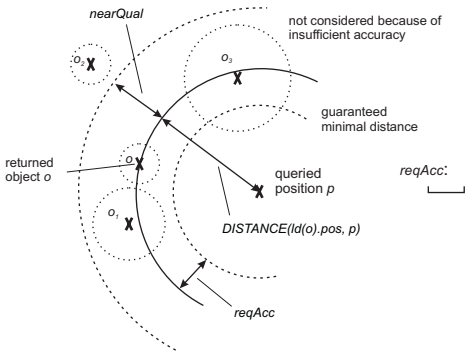

Figure 4: Scenario of a nearest neighbor query.

1. Again assuming that the probability of being at a certain location is uniformly distributed over the object's location area.

4 Architecture of the Location Service

In this section, we propose an architecture for a large-scale location service together with the basic algorithms for registration, handover and query processing. An outline of possible optimizations will be given in Section 6.5.

A LS as part of a future location-aware information system for a larger city may have hundred thousands of users. To ensure the scalability required for such a large-scale deployment of the service, the location servers that belong to the LS are organized in a hierarchical manner. Similar hierarchical architectures have been proposed for the location management in Personal Communications Services [22] and the GLOBE location service for mobile software objects [20].

A location service (LS) is configured to cover a certain geographic area, its so-called *service area*. Only objects that are located within its service area can be registered with the LS to be tracked. Tracked objects that move out of the service area are automatically deregistered.

The service area covered by an LS is structured in a hierarchical fashion: A service area can be subdivided into sub service areas, which again can be subdivided, and so on. We denote the sub areas of a given service area to be *child service areas* of this *parent service area*. The service area covered by the entire LS is called the *root service area*. While the shape of a service area may be any kind of polygon, the following two requirements must be fulfilled: (1) A non-leaf service area consists of their child service areas, that is, it is the union of its child areas, and (2) sibling service areas do not overlap.

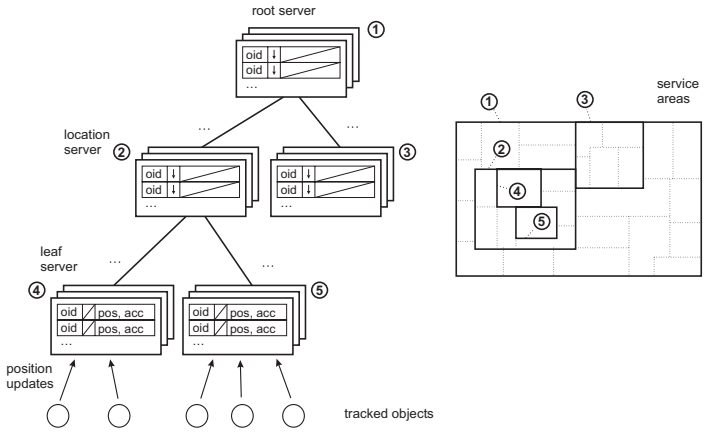

Figure 5: Basic architecture of the LS.

Associated with each service area is a location server, which is responsible for tracking all the objects visiting its service area. Consequently, the service area hierarchy resembles the location server hierarchy. The *root server* is the only server in the hierarchy without a *parent* server. Those servers that do not have *children* in the server hierarchy are denoted as *leaf servers*, all other servers are *non-leaf servers*. Fig. 5 shows an example for a part of a server hierarchy with the corresponding service areas.

- ***Leaf servers*** are associated with leaf service areas only. Since leaf service areas are not supposed to overlap, at any point in time for each tracked object there exists exactly one leaf server that is responsible for keeping track of the object's position. We will call this leaf server the object's *agent*. Of course, whenever a tracked object moves from one service area to another, the tracking responsibility has to be handed over to another leaf location server, which then becomes the object's new agent.
- A ***non-leaf server*** is responsible for a service area that is the union of the service areas associated with its child servers. Obviously, the size of service areas associated with the servers increases from server to server in a leaf-to-root direction. A non-leaf server records all tracked objects that are currently within its service area and for each of these objects stores a so-called *forwarding reference*, where an object's forwarding reference identifies the child server that is responsible for the child service area this object is currently located in.

Consequently, only leaf location servers store sighting records, whereas non-leaf servers store a reference to the child server that is "closer" to the corresponding sighting record. Obviously, the collection of forwarding references stored for any given tracked object in the server hierarchy specifies a path from the root server to the object's agent, storing the sighting record.

Location updates for a tracked object are always sent to the object's agent, which then updates the object's sighting record. When a tracked object moves to a new service area, the tracking responsibility is handed over and the forwarding path has to be adapted accordingly. Handover processing must then make sure that the tracked object learns about its new agent. How many servers are affected by the path update operation depends on what level of service area the handover

occurs. Of course, due to the hierarchical organization of service areas, handovers between lower level service areas will occur more rarely than for higher level areas.

Fig. 6 shows a three-layered server hierarchy, which we will use to briefly sketch how handover and query processing works in a server hierarchy. Details of the algorithms can be found in the next section. In our example, we assume that Server 4 (s_4 for short) has detected that some tracked object has moved out of its service area. In consequence, a handover request is sent to its parent server, which recognizes that the tracked object is still in its own (higher-level) service area. Therefore, instead of forwarding the request further up the hierarchy, s_2 sends it to that child whose service area includes the object's new position. Finally, s_5 sends the handover acknowledgement back to s_4, thereby updating the forwarding path, and s_4 informs the tracked object of its new agent.

In order to execute a position query the forwarding path is used to find the agent of the corresponding tracked object. In our example, we assume that the position query is issued at s_4 and the corresponding object currently resides in the service area associated with s_6. The position query is forwarded up to the root as only there a forwarding reference for the object is found. From there the request is forwarded along the object's forwarding path down to s_6, which finally sends the answer to s_4. Note that if the object had been located in the service area of s_5, the request would have been forwarded only up to s_2.

For the range query in our example, we assume that the specified area overlaps with the service areas of s_6 and s_7. A query request is propagated upwards from s_4 until a server is found, whose service area includes the specified area entirely, which is s_1. From there the request is propagated downwards to all children with service areas overlapping with the specified area, until reaching s_6 and s_7. Both leaf servers determine the objects that are in the corresponding sub area and send the result to s_4, which is responsible for constructing the answer returned to the client.

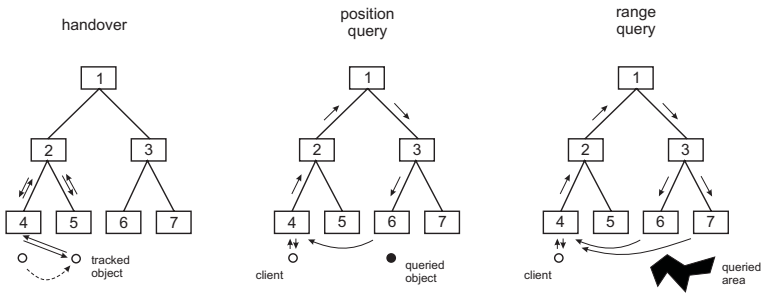

Figure 6: Basic algorithms of the LS.

The possibility to partition a service area among a number of servers is a prerequisite for a scalable location service. Due to the hierarchical organization of servers, we can gain performance by exploiting the locality of operations. Obviously, most handovers will be local since tracked objects are supposed to move to adjacent service areas. Further, we expect that also queries will show a rather high degree of locality because users of location-aware applications are typically interested in objects in their vicinity (e.g., all restaurants in walking distance, or persons that are in the same room). A high locality of queries has been observed for location-aware applications [7], for phone calls in mobile communication systems [10] and for the Globe location service [20]. An additional advantage of a hierarchical server structure is that it goes in line with hierarchically organized administrative domains, which is an important criteria for Internet-scale services (e.g., see DNS [13]).

The performance of the system is influenced by the height of the hierarchy, the fan-out of nodes (i.e., the number of children for a location server) and the size of the (leaf) service areas. The optimal setting of these parameters depend on the number of tracked objects in the root service area, their distribution over this area (e.g., where hot spots are located), and their mobility patterns (e.g., average speed, degree of locality). Moreover, the setting depends on what type of queries are requested with which frequency, that is, it depends of the mix of queries issued by the applications of the LS. Another aspect is the placement of servers, which has an impact on communication cost and latency. Due to the locality of update and query operations (leaf) servers should be close to their service areas in terms communication cost and latency.

So far, we have assumed that each service area is associated with a single server. However, due to fault-tolerance and performance reasons it is conceivable that multiple location servers are assigned to the same service area. In particular, this will be important for the root server and higher-level servers which may have to manage a large number of objects. As will be seen below, each location server maintains a database to keep track of objects visiting its service area. If there are multiple location servers associated with the same service area, this database either can be replicated on these servers or the data can be divided into partitions that are assigned to servers. For example, information about tracked objects can be partitioned based on some portion of the object id, which is similar to the partitioning of the Home Location Register in Personal Communications Services, such as GSM. Since location updates are supposed to occur very frequently, replica-

tion of sighting records would cause a significant overhead. So it has to be carefully checked which information on which level in the hierarchy can be replicated without causing prohibitive cost. A detailed discussion of these issues is beyond the scope of this paper.

5 Location Server Data Storage

Location servers need to maintain configuration information and databases that they use to keep track of objects visiting their service area. To speed up query processing a spatial index containing the position information of the tracked objects is used to find the candidates for a range or nearest neighbor query and a hash table containing object identifiers to quickly find the object belonging to a position query.

In order to maintain the server hierarchy and the associated service areas, each location server stores a configuration record on persistent storage. A configuration record *c* stored by a server, say *s*, includes the following information:

c.sa:	This component defines the service area associated with *s*.
c.parent:	This component identifies the parent server of *s*. For the root server *s.parent* is undefined (ε).
c.children:	This set includes a child record for each child of *s*. A child record encompasses two fields *id* and *sa*, which specify the child's identifier and associated service area, respectively. Clearly, for a leaf node *s.children* is empty.

Fig. 7 shows the components involved in the storage of the location data. A tracked object is called *visitor* of a location server if it is currently located in this server's service area. To keep track of its visitors, each location server additionally maintains a so-called *visitor database* (*visitorDB* for short), which includes a visitor record for each visitor. The structure of a visitor record depends on whether it is stored at a leaf or non-leaf server.

A visitor record *v* stored on a non-leaf server consist of the following components:

v.oId:	Identifier of visiting object.
v.forwardRef:	Forwarding reference to the child server, which is next on the path to the visitor's agent.

A visitor record *v* on a leaf server includes the following components:

v.oId:	Identifier of visiting object.
v.sightingRef:	Reference to the visitor's sighting record in the *sightingDB* (see below).
v.offeredAcc:	The currently offered accuracy for locating the visitor.
v.regInfo:	Registration information record for the visitor, encompassing the fields *reg*, *desAcc* and *minAcc* specifying the registering instance, the desired accuracy and the minimal accuracy, respectively.

Each leaf server in addition maintains a *sightingDB*, which includes a sighting record for each visitor. Each sighting record is associated with an expiration date, which is extended accordingly whenever the visitor contacts the location server in order to update its position. When the sighting record expires, the visitor is automatically deregistered, that is, its visitor records are removed from the entire hierarchy. That is, we adopt the soft state principle to make sure the information associated with a tracked object is eventually removed from location servers even if the object fails.

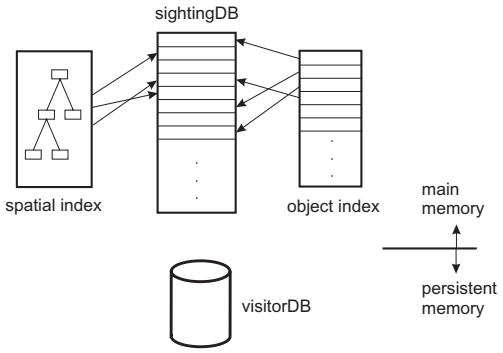

Figure 7: Data storage components.

We decided to store the *sightingDB* in volatile memory for two reasons. First, we expect position updates to occur very frequently, and hence, in particular updates should be performed most efficiently (see our measurements in Section 7). Second, the overhead for making location data recoverable from secondary storage is not worth the effort since the recorded positions would be most probably out-dated anyway after recovery. Instead, the position updates sent (periodically) by the tracked objects can be used to restore the data in volatile memory.

A spatial index over the position information in the sighting records (e.g., a Quadtree [17] or a R-Tree [6]) is used to efficiently retrieve the results for range or nearest neighbor queries. Position queries are efficiently processed by using a hash index structure defined over the object identifiers. The index structures are also stored in volatile memory. Our measurement results reported in Section 7 show that even the spatial index can be built up very quickly (e.g., as update requests arrive after system restart).

The *visitorDB* is kept in persistent storage, which is updated only when an object is registered, deregisters or a handover occurs. In other words, the objects' forwarding paths are supposed to survive system failures. Persistent registration information also allows a location server to ask a visitor for a position update to restore its position information, for example, when a position query arrives for this particular visitor after system restart.

6 Algorithms

In this section we will describe the major algorithms for implementing the LS, namely registration, handover and query processing. When describing the algorithms we will assume tracked objects with local positioning systems. However, an extension of the algorithms to also support stationary tracking sensors is straightforward.

Moreover, we will assume the existence of a mechanism that helps a client of the LS to find a leaf location server close-by. This server, which we will call *entry server*, is contacted by the client in order to access the LS. A mechanism for finding the entry server could be provided by a local lookup service, such as Jini (see [21]). A client that is at the same time a tracked object already knows the closest leaf server through the handover mechanism.

6.1 Registration

To register a mobile object with the LS, the registering instance (e.g., the mobile object itself) sends a registration request *registerReq* to its entry server (see Section 3.1 for a description of the request parameters). Then it waits for the response *registerRes* to arrive. The registration procedure is shown in Algorithm 6-1.

The first phase of the registration procedure starts when the entry server receives *registerReq*. In this phase, the request is forwarded to the leaf server that is responsible for the object according to the object's current position, specified in the sighting record *s*. If needed, the request is forwarded upwards the hierarchy until *s.pos* is in the receiving location server's service area. From there the request is forwarded downwards, until the responsible leaf server is reached.

The registration succeeds if the LS can provide an accuracy within the requested range [*desAcc*, *minAcc*]. Only in this case the second phase of registration is started, which creates the path of forwarding pointers from the leaf to the root server. To this end, a *createPath* request is forwarded upwards, until it reaches the root server. Each server receiving this request creates the corresponding visitor record in its *visitorDB*. In addition, the leaf server stores *s* in its *sightingDB*, and finally sends a *registerRes* message, back to the registering instance. This message contains the accuracy that can be offered by the leaf server (*offeredAcc*). If registration fails due to accuracy limitations, a *registerFailed* response is returned instead.

registration request	[upon receiving registration request *registerReq(s, oInfo, desAcc, minAcc, regInst)*]
	(1) **if** (mobile object is in service area of server: *s.pos* ∈ *c.sa*) **then**
	(2) **if** (server is leaf server: *c.children* = ∅) **then**
	(3) *acc* := determine maximum accuracy with which the location information can be managed by the service
	(4) **if** (*acc* ≤ *minAcc*) **then** // registration successful
	(5) **send** *createPath(s.oId)* **to** parent server *c.parent*
	(6) *visitorDB.insert(s.oId)* // create visitor record
	(7) ***with*** *visitorDB(s.oId)* ***do*** // fill visitor record
	(8) *sightingRef* := *sightingDB.insert(s); offeredAcc* := *max(acc, desAcc)*
	(9) *reginfo.desAcc* := *desAcc; reginfo.minAcc* := *minAcc*
	(10) *reginfo.reg* := *regInst*
	(11) **endwith**
	(12) **send** *registerRes(self, max(acc, desAcc))* **to** registering instance *regInst*
	(13) **else** // registration not successful
	(14) **send** *registerFailed(self, acc)* **to** registering instance *regInst*
	(15) **endif**
	(16) **else** // forward registration downwards
	(17) *child* := select *child* ∈ *c.children* with *s.pos* ∈ *child.c.sa*
	(18) **send** *registerReq(s, oInfo, desAcc, minAcc, regInst)* **to** *child*
	(19) **endif**
	(20) **else** // forward registration upwards
	(21) **send** *registerReq(s, oInfo, desAcc, minAcc, regInst)* **to** parent server *c.parent*
	(22) **endif**

Algorithm 6-1: Registration Processing.

create path	[upon receiving registration entry request *createPath(oId)* from server ls_f]
	(1) *visitorDB.insert(oId)*
	(2) *visitorDB(oId).forwardRef* := ls_f
	(3) **if** (server is not root server: *c.parent* ≠ ε) **then** // forward create path request upwards
	(4) **send** *createPath(oId)* **to** parent server *c.parent*

Algorithm 6-1: Registration Processing.

6.2 Location Updates and Handovers

As location updates are performed frequently, they are crucial in terms of performance. Therefore, various approaches have been proposed to optimize update processing. Since this paper focuses on architectural issues rather than protocol optimizations, we will confine ourselves here to a rather simple update protocol. The interested reader is referred to [24] or [15].

A tracked object continuously compares its current position - as reported by the sensor system - with the position that has been sent most recently to its agent. If these positions differ by more than the distance defined by the offered accuracy, the tracked object sends a new *updateReq* including the current sighting *s* to its agent server. The update processing performed by an agent is depicted in Algorithm 6-2.

position update	[upon receiving position update *update(s)* from tracked object *o*]
	(1) **if** (position of sighting is outside of service area: *s.pos* ∉ *c.sa*) **then** // initiate handover
	(2) **send** *handoverReq(s, visitorDB(s.oId).reginfo)* **to** parent server *c.parent*
	(3) **receive** *handoverRes*(ls_{new}, *acc)*
	(4) **send** *handoverRes*(ls_{new}, *acc)* **to** tracked object *o*
	// remove visitor and sighting record from database
	(5) *visitorDB.remove(s.oId)*
	(6) *sightingDB.remove(s)*
	(7) **else**
	(8) *sightingDB.update(s)*
	(9) **endif**

Algorithm 6-2: Processing of position updates.

handover	[upon receiving handover request *handoverReq(s, regInfo)* from server ls_f]
	(1) **if** (mobile object is in service area of server: *s.pos* ∈ *c.sa*) **then**
	(2) **if** (server is leaf server: *c.children* = ∅) **then**
	(3) *acc* := determine maximum accuracy with which the location information can be managed by the service
	(4) *visitorDB.insert(s.oId)*
	(5) *visitorDB(s.oId).offeredAcc* := *max(acc, regInfo.desAcc)*
	(6) *sightingDB.insert(s)*
	(7) **send** *handoverRes(self, max(acc, regInfo.desAcc))* **to** ls_f
	(8) **else** // forward handover downwards and create/reset forwarding pointer
	(9) *child* := select *child* ∈ *c.children* with *s.pos* ∈ *child.c.sa*
	(10) **send** *handoverReq(s, regInfo)* **to** *child*
	(11) **receive** *handoverRes*(ls_{new}, *acc)*
	(12) **if** (*visitorDB(s.oId)* = ε) **then** *visitorDB.insert(s.oId)*
	(13) *visitorDB(s.oId).forwardRef* := *child*
	(14) **send** *handoverRes*(ls_{new}, *acc)* **to** ls_f
	(15) **endif**
	(16) **else** // forward handover upwards and remove forwarding pointers
	(17) **send** *handover(s, regInfo)* **to** parent server *c.parent*
	(18) **receive** *handoverRes*(ls_{new}, *acc)*
	(19) *visitorDB.remove(s.oid)*
	(20) **send** *handoverRes*(ls_{new}, *acc)* **to** ls_f
	(21) **endif**

Algorithm 6-3: Handover processing.

An agent receiving a position update checks whether *s.pos* is still within its service area. If this is the case, it just updates the sighting record in its *sightingDB*. Otherwise, it initiates the handover procedure by sending a handover request to its parent. This message contains *s* and the registration information recorded for that object. The *handoverReq* message is propagated upwards until a receiver's service area includes *s.pos*. Starting from this receiver, the handover request if for-

warded downwards until the leaf server, whose service area contains *s.pos*, is reached. When this leaf server receives *handoverReq*, it becomes the object's new agent. That is, it creates a new visitor record in its *visitorDB*, includes *s* in its *sightingDB*, and returns a *handoverRes* message. The response includes the accuracy offered by the new agent, which is in the requested accuracy range.

Due to the hierarchical server organization, most handovers will only involve one non-leaf node. The probability that a non-leaf node is involved in handover processing decreases in a leaf-to-root direction.

The message *handoverRes* is propagated back along the path the request was sent on. The non-leaf servers receiving this message update the object's forwarding path in their *visitorDB*s accordingly, that is, remove the sub path leading to the old agent and establish a new sub path to the new one. Finally, the old agent informs the tracked object of its new agent and removes its entries from both *visitorDB* and *sightingDB*.

The processing of a handover request is shown in Algorithm 6-3.

6.3 Position Query

A client of the LS issues a position query by sending a *posQueryReq* message for a tracked object *o* to an entry server (see Algorithm 6-4). If the entry server itself stores *o*'s visitor record, it accesses its *sightingDB* to retrieve *o's* position. Otherwise, the entry server forwards the request to its parent, and then waits for the answer to arrive. Once *o*'s position is available, the entry server returns the corresponding location descriptor in a *posQueryRes* message to the client.

A non-leaf location server receiving a forwarded position query sends this message to its parent, if it does not itself store a visitor record for *o*. Otherwise, it forwards the request downwards following the forwarding pointer recorded in *o*'s visitor record. When a leaf server receives a forwarded position query from its parent, it retrieves *o*'s position from its *sightingDB* and returns the corresponding location descriptor to the entry server, whose address has been forwarded along with the query.

In summary, query processing starts at a leaf sever, which forwards the request up to the first server storing the corresponding visitor record, if needed. An alternative approach would have been to send the request immediately to the (partitioned) root. However, we decided against this alternative since we assume that clients are more interested in objects in their vicinity (see also our discussion on locality in Section 4) and hence the distance to the node storing the position information is on average shorter from a leaf server than from the root.

position query	[upon receiving position query *posQueryReq(oId)* from client *o*] (1) **if** (queried object is managed by server: *visitorDB(oId)* $\neq \varepsilon$) **then** (2) *ld* := **new** location descripor // create and return location descriptor (3) *ld.pos := sightingDB.objectHash(oId).pos* (4) *ld.acc* := *visitorDB(oId).offeredAcc* (5) **else** // forward query upwards (6) **send** *posQueryFwd(oId, self)* **to** parent server *c.parent* (7) **receive** position query results, *posQueryRes(ld)* (8) **endif** (9) **send** *posQueryRes(ld)* **to** client *o*
position query fwd	[upon receiving position query forward *posQueryFwd(oId, ls_e)*] (1) **if** (queried object is managed by server: *visitorDB(oId)* $\neq \varepsilon \wedge$ *c.children* $= \emptyset$) **then** (2) *ld* := **new** location descripor // create and return location descriptor (3) *ld.pos* := *sightingDB.objectHash(oId).pos* (4) *ld.acc* := *visitorDB(oId).offeredAcc* (5) **send** *posQueryRes(ld)* **to** entry server *ls_e* (6) **elseif** (server has forwarding pointer: *visitorDB(oId)* $\neq \varepsilon$) **then** // forward packet downwards (7) **send** *posQueryFwd(oId, ls_e)* **to** child server *visitorDB(oId).forwardRef* (8) **else** // forward packet upwards (9) **send** *posQueryFwd(oId, ls_e)* **to** parent server *c.parent* (10) **endif**

Algorithm 6-4: Processing of position queries.

6.4 Range Query

A client can issue a range query by sending a *rangeQueryReq* to an entry server (see Algorithm 6-5). If multiple leaf servers are involved in query processing, these servers return their partial results to the entry server, which is responsible for collecting them in *objects*. A returned partial result in *rangeQuerySubRes* consists of a specification of the portion of the specified area that is covered by the sender plus a set of (identifier, location descriptor)-pairs, one for each tracked

object that qualifies for this part of the area. Variable *covered* is used for checking whether or not all partial results have already been received.

If the entry server's service area does not cover the specified area entirely, the range query request is forwarded in a leaf-to-root direction until a server is reached which covers the area completely. Then the query is forwarded downwards to all leaf servers of the corresponding subhierarchy whose service areas overlap with the specified area. Each of these leaf servers uses the spatial index to identify the tracked objects qualifying for the query and retrieves their sighting records from its *sightingDB*. Finally, the result is sent to the entry server. After having received all partial results, the entry server returns the outcome of the query in a *rangeQueryRes* message to the client.

Before comparing the specified area with the service area of a location server, it is enlarged at all sides by a margin of *reqAcc* (using the function *Enlarge*). Otherwise, a server who is the agent of possible candidates for the range query, might by missed, as they can be outside of the specified area (see Section 3.2).

Obviously, the cost of processing a query depends on the number of leaf servers involved, which depends on the size of the specified area. In order to control the cost of range queries it is conceivable to limit the size of the area that applications are allowed to specify in their requests.

range query	[upon receiving range query *rangeQueryReq(area, reqAcc, reqOverlap)* from client *o*]
	(1) *objects* := ∅ // for collecting the results
	(2) *covered* := ∅ // for checking if all results have been received
	(3) **if** (*area* overlaps with service area of the server: *Enlarge(area, reqAcc)* ∩ *c.sa* ≠ ∅) **then**
	// get appropriate location descriptors from sighting database using spatial index
	(4) **foreach** *s* **in** *sightingDB.spatialIndex.objectsInArea(area, reqAcc, reqOverlap)* **do**
	(5) *objects.append({ s.oId,* **new** *ld(s.pos, visitorDB(s.oId).offeredAcc) })*
	(6) *covered* := *area* ∩ *c.sa*
	(7) **endif**
	(8) **if** (part of *area* lies outside service area: *Enlarge(area, reqAcc)* – *c.sa* ≠ ∅) **then**
	// forward query upwards
	(9) **send** *rangeQueryFwd(area, reqAcc, reqOverlap, self)* **to** parent server *c.parent*
	(10) **until** area is entirely covered: *covered* = *area*
	(11) **receive** range query results, *rangeQuerySubRes(objs, a)*
	(12) *objects* := *objects* ∪ *objs; covered* := *covered* ∪ *a*
	(13) **end**
	(14) **endif**
	(15) **send** *rangeQueryRes(objects)* **to** client *o*

range query fwd	[upon receiving range query forward *rangeQueryFwd(area, reqAcc, reqOverlap, ls_e)* from forwarding server *ls_f*] (1) **if** (area overlaps with service area of the server: *Enlarge(area, reqAcc)* ∩ *c.sa* ≠ ∅) **then** (2) **if** (server is leaf server: *c.children* = ∅) **then** // get appropriate location descripors from sighting database using spatial index (3) *objects* := ∅ (4) **foreach** *s* **in** *sightingDB.spatialIndex.objectsInArea(area, reqAcc, reqOverlap)* **do** (5) *objects.append({ s.oId,* **new** *ld(s.pos, visitorDB(s.oId).offeredAcc) })* (6) **send** *rangeQuerySubRes(objects, area* ∩ *c.sa)* **to** entry server *ls_e* (7) **else** // forward query downwards (8) **foreach** *child* **in** *c.children* **do** (9) **if** (area overlaps with child service area: *Enlarge(area, reqAcc)* ∩ *child.c.sa* ≠ ∅ **and** child was not forwarding server: *child* ≠ *ls_f*) **then** (10) **send** *rangeQueryFwd(area, reqAcc, reqOverlap, ls_e)* **to** *child* (11) **endif** (12) **endif** (13) **if** (part of area lies outside service area: *Enlarge(area, reqAcc)* – *c.sa* ≠ ∅ **and** parent was not forwarding server: *c.parent* ≠ *ls_f)* **then** // forward query upwards (14) **send** *rangeQueryFwd(area, reqAcc, reqOverlap, ls_e)* **to** parent server *c.parent* (15) **endif**

Algorithm 6-5: Processing of range queries.

6.5 Caching

Many hierarchically organized information systems employ caching mechanisms particularly to reduce the overhead caused by tree traversals (e.g., see [20] or [13]). The Domain Name System, for example, relies to a great deal on caching to achieve its scalability and performance. Our following discussion will concentrate on further optimizations by means of caching performed on leaf location severs. Clearly, similar caching mechanisms can be used on the clients of the LS. Caching mechanisms on non-leaf servers will be the focus of future work.

The following information may be cached on leaf servers:

(Leaf server, service area): The efficiency of handovers and range queries can be enhanced by caching for each locally known leaf server the associated service area. To spread this mapping information, in each request and response message forwarded within the server hierarchy the originator of the message includes a specification of its (leaf) service area. A leaf server receiving such a message stores the included mapping in its cache. Using this mechanism, leaf server will soon learn about the service areas of other leaf servers, especially those of its neighbors. When executing a handover or a range query concerning a certain area, the entry server first checks whether it can determine the leaf server(s) for this area from its cache. If this is the case, it can contact the server(s) directly, without traversing the hierarchy. As the service areas of location servers are assumed to change seldomly, the probability of out-dated cache entries will be low. Therefore, we expect the performance improvements achieved by this mechanism be worth the additional (little) overhead in almost all cases.

(Tracked object, current agent): To speed up position queries, the address of a tracked object's current agent can be cached. Upon receipt of a response for a query, a leaf server stores a cache entry with address of the response's originator for each object identifier included in the response. When an entry server receives a position query for a tracked object, it first checks whether it finds an entry for the corresponding object in its cache. If so, it contacts the recorded leaf server directly. However, for highly mobile objects cache entries may become out-dated soon, depending on the size of the service areas as well as the speed and moving pattern of the objects. Consequently, how well this mechanism performs depends on the frequency of handovers between leaf service areas compared with the frequency of position updates. If agent information becomes obsolete too fast it has been suggested to use cache pointers to higher-level servers to shorten the search path (see for example [1]). The service areas associated with those servers are larger, and hence handovers on a higher level are not that frequent as on the leaf level.

(Tracked object, position descriptor): Besides storing information about the current agent of a tracked object, it is also possible to cache the position descriptor returned as result for a previous query. When the location server receives another query concerning the same object(s), it can retrieve the position information directly from the cache, provided the information is still accurate enough. (this can be determined based on the object's maximum speed with the estimation method mentioned in Section 3). Whether or not caching of position information for an object makes sense depends on various factors: The speed of this object, the frequency of position queries, as well as the position accuracy requested for this object.

7 Experiments

A first prototype of the LS, implemented in Java 1.2, is available. It provides the basic functionality for position updates, handover and query processing. However, the caching mechanisms described in Section 6.5 are not included yet. Therefore, all measurement results reported below show the performance of the system without caching.

7.1 Data Storage

As pointed out in Section 5, we decided to store position information in a main memory database. For the spatial index we used a Point Quadtree implementation [17], which we found to be very well suited for our purpose. Persistent storage is provided by a DB2 database, which is accessed via JDBC.

To evaluate the performance of our main memory solution, we have measured the throughput of the data storage on a SUN Ultra machine with a 450 MHz CPU and 1 GB of memory, where the load was generated locally. The results of these measurements are depicted in Table 1 for the different types of operations. In our experiment, we started with an empty location server associated with a service area of 10 by 10 km. Then we consecutively registering 25,000 tracked objects with random positions in this area. Subsequently, 10,000 position updates and 10,000 position queries were consecutively performed for randomly selected objects, and finally 10,000 range queries for random areas of three different sizes were executed.

The results show that the spatial index can be built-up very fast (in our case 25,000 inserts in about 1 second), which is in particular important for crash recovery. Remember that the spatial index is lost due to failures and has to be built up again after system restart as position update requests come in. The main memory solution also provides for very efficient position updates. Position queries, which only involve the hash table of object identifiers, can be processed even faster. Obvi-

ously, the throughput for range queries depends on the size of the area, which determines the portion of the spatial index to be searched. Our measurements show, that the data storage can process more than 1000 queries per second even for big areas. Moreover, these results also show that data storage is not going to become a performance bottleneck in our system.

operation	operations per second
creating index	24,015 1/s
position updates	41,494 1/s
position query	384,615 1/s
range query (10 m $\times$ 10 m)	21,834 1/s
range query (100 m $\times$ 100 m)	18,450 1/s
range query (1 km $\times$ 1 km)	1,813 1/s

Table 1: Throughput of the data storage component based on a service area of 10 km $\times$ 10 km and 25,000 tracked objects.

7.2 Local and Distributed Queries

To determine the overall performance of the LS for local as well as for distributed queries, we have set up a small testing environment with 5 SUN Ultra workstations with 450 MHz CPUs and between 256 MB and 1 GB of memory connected through a 100 MBit Ethernet network. One of these machines is configured as root server with the other four as its children. Each child server is responsible for a quarter of the entire service area of 1.5 by 1.5 km (see Fig. 8). Our communication protocols are implemented on top of UDP to achieve efficient client/server and server/server interactions.

For this configuration of the LS, we have measured the response time as well as the overall throughput for position updates, position queries and range queries. To this end, we have registered 10,000 mobile objects at random positions with the LS. Three additional machines act as load generators and run parallel client processes sending the update and query requests as fast as possible to the four leaf location servers, so that each of these servers receives an equal share of the load. The queried area in range queries has a medium size of 50 by 50 meters.

Because we wanted to determine the effect of having to traverse the hierarchy, we have distinguished between local and remote queries. For position updates (which are always local in our architecture), for local position and range queries, the requests are always sent directly to the queried object's agent or the server whose service area includes the queried area. For remote queries, we have chosen another entry server.

We have also looked at the overhead caused by remote range queries that involve more than one leaf server by querying an area that is (1) included completely in the service area of one leaf server, that (2) overlaps with the service areas of two leaf servers, and (3) overlaps with the service areas of all four leaf servers. Table 2 shows the results we have obtained for our test configuration.

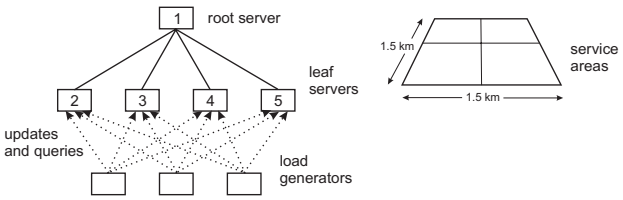

Figure 8: Testing environment for the distributed mechanisms of the LS.

The update rate achieved for this configuration would be sufficient to support position information for as much as 100,000 mobile objects with an average speed of 3 km/h and an accuracy of 25 m. Similarly, local queries can be performed very efficiently, where the performance of position query is better, mainly because of the larger results of range queries. Although still good, remote queries and especially remote range queries are more expensive since one query involves at least three location servers. Here, the optimizations proposed in Section 6.5 should definitely bring an improvement. Note that our decision to collect the range query results at the entry server before sending them back to the client causes an extra hop of communication, whose effect is considered in our measurements.

operation	response time	overall throughput
position updates	1.2 ms (with ACK)	4,954 1/s
local position query	2.0 ms	2,809 1/s

Table 2: Response time and overall throughput for different types of operations performed on the test configuration of the LS.

operation	response time	overall throughput
remote position query	6.3 ms	728 1/s
local range query	5.1 ms	1,927 1/s
remote range query (1 server)	13.0 ms	588 1/s
remote range query (2 servers)	14.6 ms	364 1/s
remote range query (4 servers)	13.8 ms	284 1/s

Table 2: Response time and overall throughput for different types of operations performed on the test configuration of the LS.

8 Conclusion and Future Work

In this paper we have presented the design of a distributed LS, which is suitable for large-scale deployment and frequent updating of the location information. Besides position queries, the service is also able to efficiently carry out distributed range and nearest neighbor queries. Its data storage component, which combines a main memory and a stable storage part, is designed to allow an efficient processing of local updates and queries. In the paper, we have described the service model and the hierarchical distributed architecture of this service as well as its data storage component and algorithms. A performance evaluation of a first prototype of the LS shows the feasibility of our concepts.

For future work, we will soon have finished with the implementation of the distributed mechanisms for the LS and continue to evaluate it in more detail. We will be looking into the influence of movement and querying characteristics on the performance of different configurations of the LS. Parameters of interest are, for example, the density of the tracked objects or their moving patterns as well as the concrete mix of different types of queries and their degree of locality. Those will be considered in comparison with the size of service areas or the height and fan-out of the location server hierarchy. Based on the results of this evaluation we will look at caching mechanisms, some of which have been outlined in Section 6.5, with the goal of further improving the scalability of the service. We will then integrate promising caching mechanisms into the LS and evaluate their influence on its performance for different usage scenarios. Further areas of research include the integration of an event mechanism as sketched in Section 1.

References

[1] A. Baggio, G. Ballintijn and M. v. Steen: Mechanisms for Effective Caching in the Globe Location Service, *Proceedings of the Ninth ACM SIGOPS European Workshop*, pp. 55-60, 2000.

[2] M. Bauer: *Event-Management für mobile Benutzer*, Master-Thesis No. 1836, Faculty of Computer Science, University of Stuttgart. In German.

[3] Keith Cheverst, Nigel Davies, Keith Mitchell and Adrian Friday: Experiences of Developing and Deploying a Context-Aware Tourist Guide: The GUIDE Project, *Proceedings of the Sixth International Conference on Mobile Computing and Networking (MobiCom 2000)*, pp. 20-31, 2000.

[4] G. Fitzmaurice: Situated Information Spaces and Spatially Aware Palmtop Computers, *Communications of the ACM*, 36(7), pp. 38-49, 1993.

[5] L. Forlizzi, R. H. Güting, E. Nardelli and M. Schneider: A Data Model and Data Structures for Moving Objects Databases. *Proceedings of the ACM SIGMOD Conference*, pp. 319-330, 2000.

[6] A. Guttmann: R-Trees: A Dynamic Index Structure for Spatial Searching, *Proceedings of the 13th ACM SIGMOD Conference*, pp. 47-57, 1984.

[7] A. Harter and A. Hopper: A Distributed Location System for the Active Office, *IEEE Network*, 36(1), pp. 62-70, 1994.

[8] A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster: The Anatomy of a Context-Aware Application, *Proceedings of the Fifth International Conference on Mobile Computing and Networking (MobiCom '99)*, pp. 59-68, 1999.

[9] J. S. M. Ho and I. F. Akyildiz: Dynamic Hierarchical Database Architecture for Location Management in PCS Networks, *IEEE/ACM Transactions on Networking*, 5(5), 1997.

[10] D. Lam, Y. Cui, D. C. Cox, and J. Widom: A Location Management Technique to Support Lifelong Numbering in Personal Communications Services, *ACM Mobile Computing and Communications Review*, 2(1), pp. 27-35, 1998.

[11] U. Leonhardt: *Supporting Location-Awareness in Open Distributed Systems*, PhD-thesis, Imperial College of Science, Technology and Medicine, University of London, 1998.

[12] H. Maaß: Location-Aware Mobile Applications based on Directory Services, *Proceedings of the Third International Conference on Mobile Computing and Networking (MobiCom '97)*, pp. 22-33, 1997.

[13] P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Proceedings of the SIGCOMM '88 Symposium*, pp. 123-133, 1988.

[14] M. Mouly and M.-B. Pautet: *The GSM System for Mobile Communications*, Palaiseau, France, 1992.

[15] N. N.: *A Comparison of Protocols for Updating Location Information*, Technical Report TR-2000-05, 2000.

[16] National Imagery and Mapping Agency: *DoD World Geodetic System 1984, its Definition and Relationship with Local Geodetic Systems*, National Imagery and Mapping Agency, 8350.2, Third Edition, 1997.

[17] H. Samet: *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990, ISBN 0-201-50255-0.

[18] Sentient Computing Project Home Page, AT & T Laboratories Cambridge, `http://www.uk.research.att.com/spirit/`, Work in Progress.

[19] M. Spreitzer and M. Theimer: Providing Location Information in a Ubiquitous Computing Environment, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pp. 270-283, 1993.

[20] M. v. Steen, F. Hauck, P. Homburg and A. Tanenbaum: Locating Objects in Wide-Area Systems, *IEEE Communications Magazine*, pp. 2-7, 1998.

[21] *Sun's Jini Homepage*, `www.sun.com/jini`, Work in Progress.

[22] J. Wang: A Fully Distributed Location Registration Strategy for Universal Personal Communcation Systems, *IEEE Journal on Selected Areas in Communications*, 11(6), pp. 850-860, 1993.

[23] R. Want, A. Hopper, V. Falcao and J. Gibbons: The Active Badge Location System, *ACM Transactions on Information Systems*, 10(1), pp. 91-102, 1992.

[24] O. Wolfson, A. P. Sistla, S. Chamberlain and Y. Yesha: Updating and Querying Databases that Track Mobile Units, *Distributed and Parallel Databases Journal*, 7(3), pp. 1-31, Kluwer Academic Publishers, 1999.