Simple Crash Recovery in a Wide-Area Location Service

G. C. Ballintijn
Fac. of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

M. van Steen
Fac. of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

A. S. Tanenbaum
Fac. of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

Abstract

We are building a wide-area location service that tracks the current location of mobile objects. The location service is distributed over multiple nodes to support 10^{12} objects on a worldwide scale. Changing information in the location service usually involves multiple nodes. If any of these nodes crashes while the information is modified, information can be lost and the location service can become inconsistent. To recover the lost information and resolve these inconsistencies, we invented a crash recovery method. The method consists of executing lost operations a second time. We show that if we focus on creating a new *consistent* state, instead of completely *restoring* the state before the crash, recovery becomes simple and efficient.

Keywords: crash recovery, fault-tolerance, wide-area distributed systems.

1 Introduction

Mobility has become increasingly prominent in information networks [1]. We use the term **mobile object** to refer to a hardware or software component in a network that changes its location. Since the location of a mobile object may change often, an efficient way is needed to obtain its current location. A **location service** is a directory service that is designed to track the location of mobile objects, and provide this information efficiently.

A location service in a wide-area network is likely to be distributed across multiple nodes. Updating the information on an object might therefore involve several nodes. This makes a global modification algorithm susceptible to node crashes, which in turn could introduce *inconsistencies* between those nodes.

As part of our research on a worldwide distributed system called Globe [2], we are building a wide-area location service. The global modification algorithm used by our location service always succeeds in modifying the (distributed) state, even if one or more of the nodes involved in the modification crash. In this paper, we show that by designing our service to use operations that are idempotent, commutative, and atomic, we can easily solve consistency problems due to node crashes. In particular, crash recovery generally does not require extra accesses to disk, and only a little extra communication is needed in the recovery phase. The focus of this paper is on dealing with inconsistent distributed state; we intend to use existing techniques to deal with media failures and other problems with persistent storage.

The rest of this paper is organized as follows. Section 2 provides an overview of our location service. Section 3 describes how location information is updated in the location service. Section 4 then explains how the update algorithms deal with node crashes. Section 5 compares our approach to the work of others. We draw our conclusions in Section 6.

2 A Wide-Area Location Service

Globe uses **object handles** to refer to objects. An object handle refers to an object throughout the object's entire life time. It is location independent, and therefore does not change when the designated object moves to a different location or when the object is replicated. However, we need to know an object's **contact address** to communicate with it. Since objects are often replicated to increase availability and performance, a single object can have multiple contact addresses at the same time, one for each replica. A traditional naming service can be used to bind human friendly names to object handles.

Our location service is used to map an object handle to one or more contact addresses [3]. Our current design goal is to support 10^{12} objects owned by 10^9 users, and still allow large numbers of update and look-up requests to be handled. Since the location service is distributed across a wide-area network, it should deal gracefully with node failures, network partitions, and (long) communication delays.

To efficiently update and look-up contact addresses, we organize the underlying wide-area network as a hierarchy of **domains**, similar to the organization of DNS. For example, a lowest level domain may represent a campus-wide network of a university, whereas the next higher level domain represents the city where that campus is located. Each domain is represented in the location service by a **directory node**. Together the directory nodes form a worldwide search tree.

A directory node has a **contact record** for every object in its domain. A contact record contains a number of **contact fields**, one for each child node. A contact field stores either a **forwarding pointer** or the actual contact addresses. A stored contact address corresponds to a contact point in the child's domain. A forwarding pointer indicates that contact addresses can be found at the child node. The contact addresses in the contact field are stored in a set. Every contact address is therefore unique in the contact field. A leaf node has only one contact field storing the contact addresses from the leaf domain.

Every contact address has a path of forwarding pointers from the root down, pointing to it. We can always locate a contact address by following this path. In the normal case, contact addresses are stored in the leaf node. However, storing addresses at higher level nodes may, in the case of high mobility, lead to considerably more efficient look-ups, as we explain in [3].

Figure 1 shows an example of a tree for one specific object. In this example, leaf node N3 stores contact addresses. A path of forwarding pointers therefore exists from root node N0, via N1, to leaf node N3. Node N1 stores, besides the forwarding pointer, also addresses from the domain of leaf node N2.

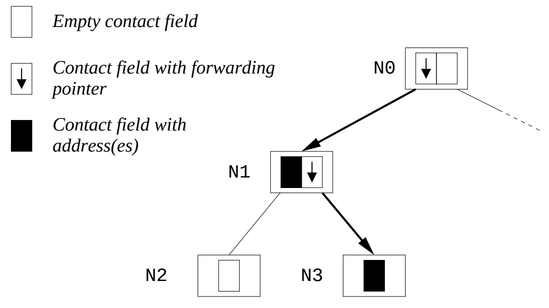
Empty contact field
Contact field with forwarding pointer
Contact field with address(es)
N0
N1
N2
N3

Figure 1: The organization of contact records in the tree for a specific object

To ensure the proper operation of the location service, the distributed search tree must adhere to certain consistency rules. Every update operation must transform a consistent tree into a new consistent tree. For our discussion here, the following rules must be obeyed:

1. A contact field must never store both a contact address and a forwarding pointer.
2. Every forwarding pointer must point to a child node which stores contact addresses or forwarding pointers.
3. If a node stores contact addresses or forwarding pointers, its parent must store a forwarding pointer pointing to that node.

The implication of the first and third rule is that on a path from the root to a leaf there is only one node where contact addresses are stored. The third rule ensures that a contact address can always be found by following a path of forwarding pointers. Both implications apply, however, only if no update operations are in progress in the tree.

Communication between between nodes, and between clients and leaf nodes is based on Remote Procedure Calls (RPC). We use a special form of concurrent RPCs that allows new operations to be started while waiting for the current operation to finish. The execution of an RPC usually involves performing another RPC at the parent, which leads to a **chain of RPCs** from the leaf upwards, possibly to the root. Problems arise when any node in the chain of RPCs crashes. The update operation will have been performed only by some nodes, which produces inconsistencies between nodes.

The focus of this paper is how our location service deals with this problem. It shows how global update operations continue after a node crash, and upon completion leave the tree in a consistent state with the operation performed. Look-up operations do not affect the consistency of the tree. We have therefore only to ensure progress for look-up operations in the face of broken links and node crashes. In practice, this problem is easy to solve, for which reason we omit further discussion.

3 Update Operations

To explain the crash recovery of update operations in the location service, we first need an understanding of the update operations themselves.

3.1 Update Operation Structure

The insert operation is illustrated in Figure 2 and 3. It consists of an upward phase in which the proper level to store the contact address is decided (Figure 2), and a downward phase in which the path of forwarding pointers is created and the contact address is inserted (Figure 3). Deciding where to store the contact address is itself distributed over the nodes on the path from the leaf to the root. The general mechanism is that a node decides for itself whether it should store the contact address, and asks its parent to agree. The parent then agrees with or overturns the node's decision.

The upward phase starts at the leaf node of the domain to which the contact address belongs. The leaf node makes a preliminary decision whether it wants to store the contact address, and informs its parent of its decision. This process is recursively repeated at every node on the path to the root. An intermediate node makes its preliminary decision based upon on its child's request and its own desire to store the contact address. The recursion stops as soon as a node is reached that already stores contact addresses or forwarding pointers, or otherwise at the root.

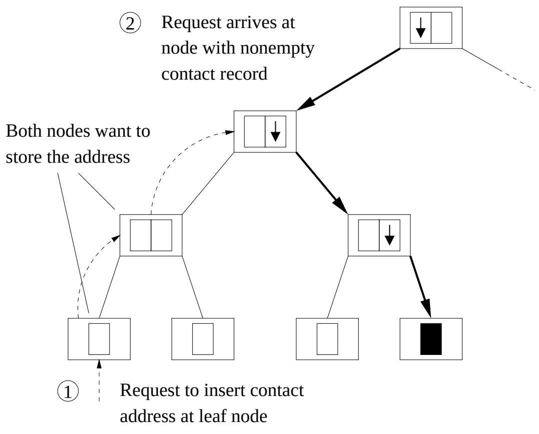
② Request arrives at node with nonempty contact record
Both nodes want to store the address
① Request to insert contact address at leaf node

Figure 2: Insert operation upward phase

The downward phase starts at the node at which the recursion stopped. This node stores either the contact address or a forwarding pointer, and informs its child what was stored. The child node decides what to do itself using its parent's reply and its own preliminary decision. If the contact address was stored higher up in the tree, the child stores nothing. If the child wants to store the contact address itself, it inserts the contact address. Otherwise, the contact address is to be stored in the subtree rooted by the child's child, and the child inserts a forwarding pointer. The child node then, in turn, informs its own child what was stored. This process is repeated at every node on the path to the leaf. The insert operation is completed after the leaf node has performed its action.

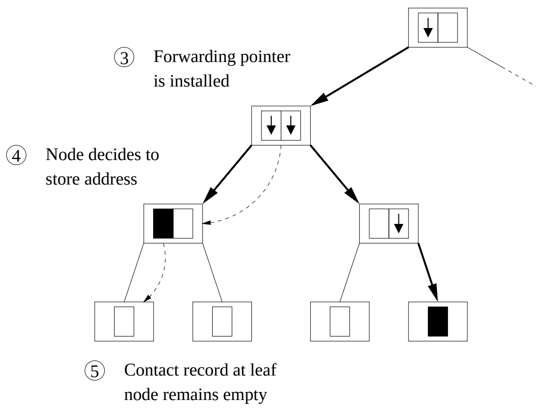
③ Forwarding pointer is installed
④ Node decides to store address
⑤ Contact record at leaf node remains empty

Figure 3: Insert operation downward phase

The delete operation consists of finding the node where the contact address is stored, deleting the contact address, and deleting the path of forwarding pointers. A delete operation starts at the leaf node of the basic domain where the contact address was inserted. It searches the contact address at the nodes on the path to the root. When the delete operation finds the contact address, it removes the contact address from the contact record. If the contact record no longer contains contact addresses or forwarding pointers, the path of forwarding pointers to it is recursively removed upwards. The delete operation is completed after a node is reached that also contains other contact addresses or forwarding pointers, or otherwise at the root.

Each update operation at a node consists of three phases.

1. Modify the contact record in main memory only. This modification is called **tentative**.
2. Inform the parent of the modification. This allows the parent to make its own modifications.
3. Make the modification permanent or discard it. Making the modification permanent or discarding it, makes the contact record **authoritative**.

Only the insert operation decides whether to make its modification permanent or to discard it. The delete operation always makes its modification permanent. The node informs the parent of its tentative modification by requesting the parent to insert either a forwarding pointer or the contact address. However, if the parent decides to store the contact address itself, the node is forced to discard its tentative modification in the third phase.

After the third phase, the node sends the RPC reply which locally completes the operation. However, we need not wait until an update operation is completed before making the tentative result available to look-up operations. The modification made in the first phase concerning the contact address is valid; it may just be tentatively stored at the wrong level.

3.2 Concurrency

A node can receive concurrently multiple update requests from its children. If these requests use different object handles, the requests can be handled independently, as every object handle has its own contact record independent of all other object handles. However, if requests use the same object handle, some form of mutual exclusion is needed. The location service uses a special form of mutual exclusion, described below. The method behaves differently, depending on whether the requests came from the same child or from different children.

If the update requests came from the same child node, the child node sent them in some specific order. This order needs to be maintained for the tree to remain consistent. If, for example, a delete is followed by an insert request, the result is (generally) that a forwarding pointer is inserted. Changing the order would result in inserting a forwarding pointer which is then immediately deleted, possibly violating the third consistency rule.

Nodes maintain the order of updates by adding a sequence number to every RPC request. The receiving node schedules the requested operations in sequence. During the execution of an operation, the operation holds a lock on the contact record used. The operation, however, is required to release this lock when it blocks while performing an RPC. This allows the next operation to be run. When the RPC is finished, the blocked operation can continue after acquiring the lock again. Operations continue after an RPC in the same order as they started the operation. The operations thus run concurrently in a **pipelined** fashion.

If the update requests came from different child nodes, a random ordering is used. This does not affect consistency, since the update operations modify different contact fields. It can however influence the final state of the tree, as shown in Figure 4.

3.3 Availability

The location service is highly available in the face of network partitions and unavailable (crashed) nodes. This availability comes from its use of concurrent RPCs and the use of tentative results in update operation. If, for instance, a subtree is cut off from the rest of the search tree, operations in the subtree can still continue. Update operations are effectively queued at the root node of the subtree, while waiting for the connection to be restored. Tentative results will, however, be available for look-up operations from the subtree. Availability for look-up operations can easily be improved further by using a form of primary-backup replication analogous to DNS.

4 Crash Recovery

The purpose of the recovery mechanism is twofold: (1) it corrects the inconsistencies created by a node crash during an update operation, (2) it allows update operations to complete, even though nodes crash during the execution. The recovery mechanism must deal with lost RPC requests and replies, and with RPC chains severed when the node crashed after having sent update requests to its parent. The node loses all tentative changes, but the parent (unaware of the crash) still modifies its own contact record. The severed chain of RPCs thus results in some nodes performing their part of the update operation, while other nodes do not. This may result in inconsistencies in the tree. Figure 5 shows how a partly executed delete operation creates such an inconsistency.

4.1 Assumptions

Our model assumes a fail-silent system [4]. That is, a faulty node just stops sending messages and crashes; no erroneous messages are sent before the crash. The node is rebooted after a finite amount of time. Since the focus of this research is on resolving inconsistencies in the distributed search tree, we assume disk writes are atomic and that disks are not affected by a node crash. A node crash therefore results only in the permanent loss of main memory, and thus all tentative modifications. We intend to use existing techniques to implement this model.

4.2 Recovery Mechanism

The central notion of the recovery mechanism is that the *children* of the recovering node have all the information required for the recovery. The problem of lost messages and inconsistencies is solved by having

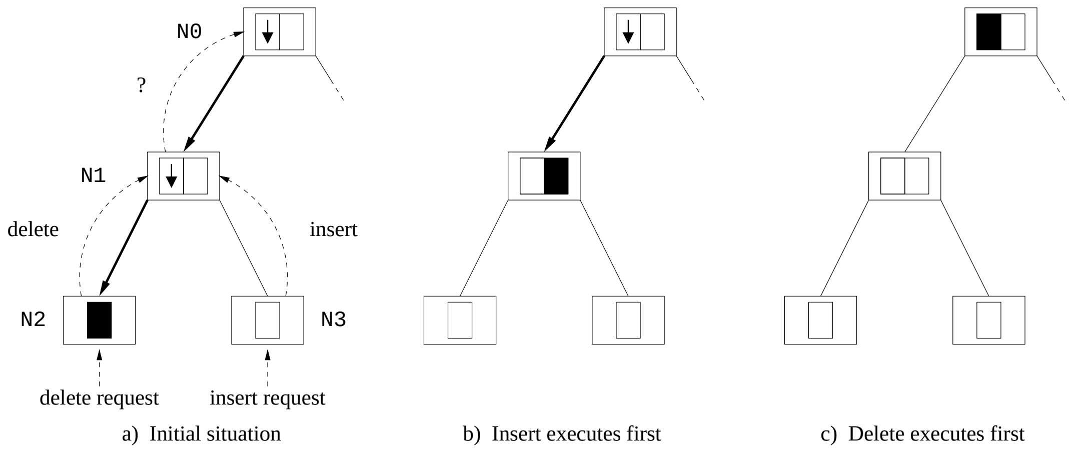

Figure 4: Race condition between a delete and an insert operation.

the children retransmit their outstanding requests and the recovering node perform the requested operations, possibly again. No separate correction phase is used. The recovery mechanism is in essence a form of sender-based message logging [5].

After the crash, the recovering node enters the **recovery phase**, and informs its children it is recovering from a crash. Each child then retransmits its outstanding requests in their original order. These requests will be executed (almost) normally by the parent, as we describe below. After all children have retransmitted their old requests, they continue with sending new requests. The recovery phase is finished when all inconsistencies have been resolved, which we also describe below.

4.3 Resolving Inconsistency

The recovering node resolves inconsistencies by simply executing the operations again that created the inconsistencies. The important thing to note is that these operations are not treated as something special. Instead, the recovering node informs its parent of its intended modification as it would normally do, and awaits the parent's answer. The parent, in turn, will use its current version of the contact record, and report back to the recovering node as usual. In other words, by simply replaying the lost operations, as if nothing had happened, we can show that the tree eventually enters a consistent state again.

How can this be? Basically, we force the recovering node and its parent to become mutually consistent again. Consider first the situation of re-executing an insert operation at the recovering node. If its associated contact field at the parent already stores contact addresses, the parent will store the new contact address, and the parent will tell the recovering node to discard its modification. If the contact field already stores a forwarding pointer, the parent will tell the recovering child to store the contact address. If the contact field is empty, the parent can choose what to do, and subsequently tell the recovering node what it should do.

Now consider the situation when the recovering node re-executes a delete operation. If the delete operation was already executed by the parent, the parent will find the associated contact field empty. In that case, the parent can simply tell the recovering node it has completed its part of the operation, as usual. If the parent did not receive the delete request before, it will simply perform the delete operation as usual.

Unfortunately, during the recovery phase the retransmitted operations have to deal with the inconsistencies between the two contact records. These inconsistencies are violations of consistency rules two (a forwarding pointer pointing to an empty child node) and three (a nonempty contact record without a forwarding pointer at the parent). Consistency rule one applies to the local contact record only, and therefore can be violated by a node crash.

The delete operation behaves correctly when faced with these inconsistencies. The correct execution of the insert operation is, however, threatened. Normally, a node decides to inform its parent of an insertion using information from its own contact record. If

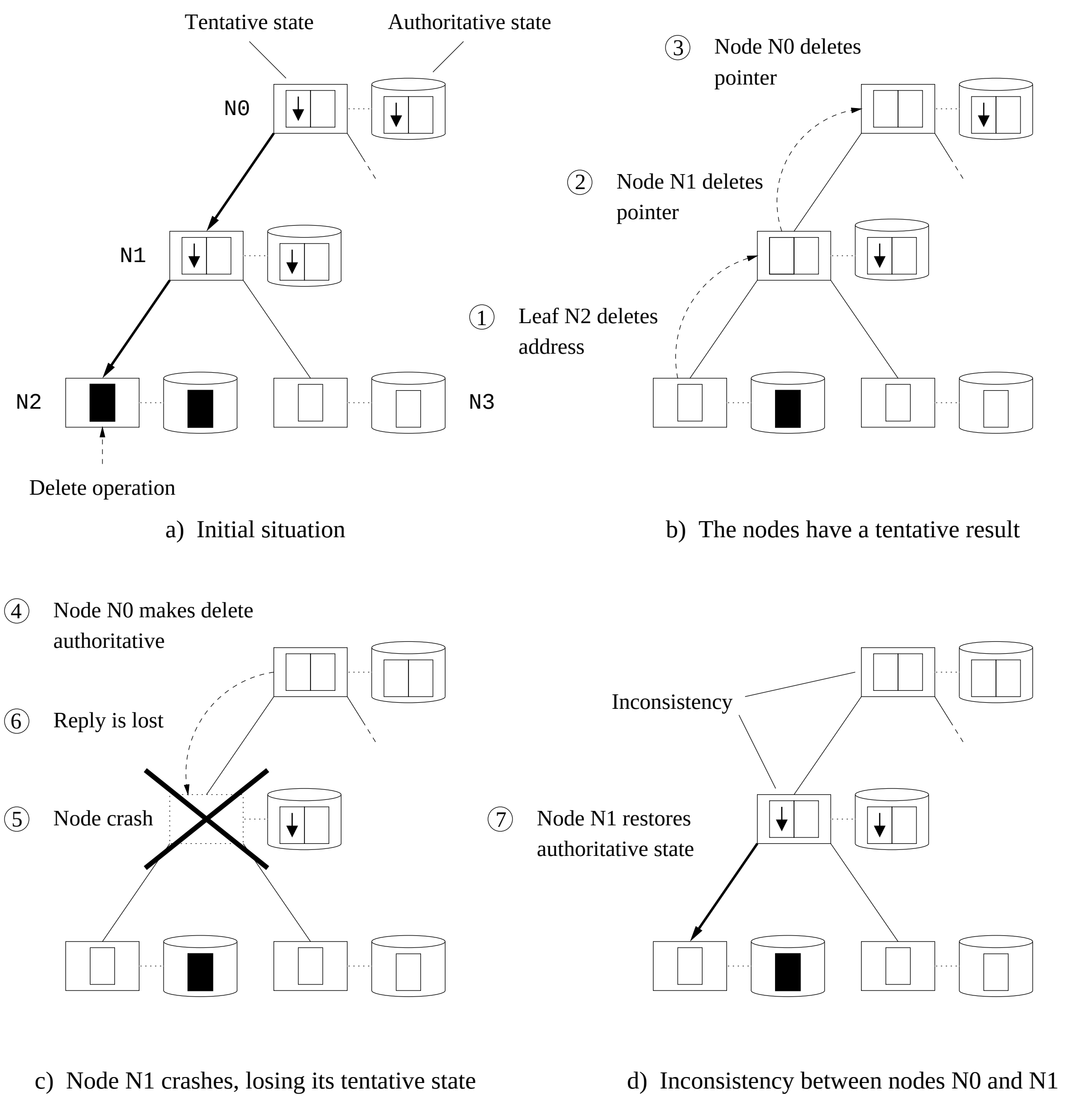
Tentative state
Authoritative state
N0
N1
N2
N3
Delete operation
a) Initial situation
③ Node N0 deletes pointer
② Node N1 deletes pointer
① Leaf N2 deletes address
b) The nodes have a tentative result
④ Node N0 makes delete authoritative
⑥ Reply is lost
⑤ Node crash
c) Node N1 crashes, losing its tentative state
Inconsistency
⑦ Node N1 restores authoritative state
d) Inconsistency between nodes N0 and N1

Figure 5: Occurrence of inconsistency

the contact record already contains contact addresses or forwarding pointers, the parent is assumed to have a forwarding pointer, and the node does not contact its parent. However, if the two contact records are not consistent (as shown in Figure 5d), the insert operation would incorrectly assume the forwarding pointer exists, and stop prematurely.

To deal with this kind of inconsistency, the recovering node behaves differently in the recovery phase in the following way: The node *always* forwards insert requests to its parent, instead of only when the node inserts the *first* contact address or forwarding pointer in the contact record. It is possible that the parent inserts the forwarding pointer for the second time, but this is not a problem given the idempotent nature of inserting a forwarding pointer: only one forwarding pointer is stored.

The recovery phase is implemented by distinguishing the RPC requests *resent* after a node crash as **recovery requests**. A child node sends a special **end-recovery-phase** message to signal the end of its recovery requests. If the recovering node has received end-recovery-phase messages from all it children, it knows that when all currently running operation are finished, its contact records are consistent with the ones at its parent.

4.4 Correctness

This scheme works for three reasons: the modification due to an update operation is made permanent **atomically**, the modification is **idempotent**, and modifications from different child nodes are **commutative**.

The modified contact record is saved to disk in an atomic fashion. The node has either saved the modified contact record in the last phase of the update operation, or it crashed before saving. Contact records on disk are therefore never corrupted by a node crash. Since a contact record is saved to disk with one atomic write operation, the recovery mechanism does not have to worry about inconsistencies within a node.

The modifications to the contact record are idempotent, since contact fields use sets to store contact addresses, and a forwarding pointer is basically a boolean value. Inserting a contact address in or deleting it from the contact field for a second time does not change the contact field. Setting or clearing the forwarding pointer a second time does not change the contact field either. Being idempotent allows the modifications to be redone without adverse effects. The idempotency also applies to groups of operations, since the update operations are retransmitted in the original order and for every contact address or forwarding pointer only the last operation (insert or delete) really matters.

The modifications requested by different child nodes are commutative, since the different children operate on different contact fields. Requests retransmitted by different child nodes therefore do not interfere. However, the final result can be different than expected before the crash, since the ordering of request of two children can be different during the recovery. This is not a problem, since the issue is not whether we restore the original state, but instead, that we end in a *consistent* state. As shown in [6] our scheme also works for multiple node crashes.

4.5 Leaf Nodes

Every leaf node has a *persistent* message log to store incoming update requests from clients of the location service. This is needed, since we do not want to place the responsibility of retransmitting lost requests on the clients of the location service. The RPC mechanism saves every incoming request to the log before handling it. When the operation is completed, the request is deleted from the log. A possible way to minimize the overhead of logging is the use of non-volatile RAM. A leaf node replays its message log during the recovery phase after a crash.

5 Conclusion

The use of sender-based message logging provides a way to easily implement crash recovery in our location service. Based on inherent properties of the system, such as idempotent and commutative set-like operations, the crash recovery basically comes for free. In contrast to a more heavy-weight general crash recovery mechanisms [7], which attempt to restore the lost state, no special correction phase is needed. The same operations that have to be executed anyway, are used for recovery. Our simple solution allows for efficient and scalable update operations. To validate our ideas we have built a prototype of our location service. Future work will consist of performing measurements on this prototype.

References

[1] G. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *Computer*, vol. 27, nr. 4, pp. 38-47, Apr. 1994.

[2] M. van Steen, P. Homburg, and A. S. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, pp. 70-78, Jan. 1999.

[3] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum, "Locating Objects in Wide-Area Systems," *IEEE Communications Magazine*, pp. 104-109, Jan. 1998.

[4] J.-C. Laprie, "Dependability – Its Attributes, Impairments and Means," In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood (ed.), *Predictably Dependable Computing Systems*, pp. 3-24, Springer-Verlag, Berlin, 1995.

[5] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Proc. of the 17th Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 14-19, Pittsburgh, PA, July 1987. IEEE.

[6] G. Ballintijn, M. van Steen, and A. S. Tanenbaum, "Lightweight Crash Recovery in a Wide-area Location Service," Technical Report IR-451, Vrije Universiteit, Amsterdam, Oct. 1998.

[7] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal, and Optimal," *IEEE Trans. on Software Engineering*, vol. 24, nr. 2, pp. 149-159, Feb. 1998.