Lightweight Crash Recovery in a Wide-area Location Service

Gerco Ballintijn
Maarten van Steen
Andrew S. Tanenbaum

Internal report IR-451
October 1998

Abstract.
We are building a wide-area location service that tracks the current location mobile objects. The location service is distributed over multiple nodes, to supported 10^{12} objects on a worldwide scale. Changing the location information 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 consist 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. To validate our ideas we have built a prototype.

Keywords: *distributed systems, naming/location service, mobile computing, worldwide scalable systems, fault tolerance, crash recovery*

vrije Universiteit

Faculty of Mathematics and Computer Science

1 Introduction

Mobility has become increasingly prominent in information networks [1, 2, 3, 4, 5]. We use the term **mobile object** to refer to a hardware or software component in a network that changes its location. A mobile object usually communicates with its user and other (mobile) objects to perform its tasks. Efficient communication with an object usually requires knowledge of 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 [6], 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; media failures or other problems with persistent storage are not addressed. The research described here is a step towards providing a fully fault-tolerant wide-area location service.

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. In Section 5 we describe the prototype we built to test our ideas. Section 6 compares our approach to the work of others. We draw our conclusions in Section 7.

2 A Wide-Area Location Service

In our model, to communicate with an object we need to know its **contact address**. For example, the contact addresses of a web server object would specify that the object can be contacted through HTTP at a specific TCP address. Objects are often replicated to increase accessibility and performance. We therefore take into account that a single object can have multiple contact addresses at the same time, one for each replica.

A traditional naming service can, in principle, be used to provide the contact addresses of mobile or replicated objects. However, to support mobility, the naming service should allow the name-to-address mapping to be updated frequently. This requirement cannot be met by current naming systems such as the Internet Domain Name System (DNS) [7] or the X.500 Directory service [8], which explicitly assume that mappings are relatively stable. Consequently, we follow a different approach.

2.1 Naming architecture

The Globe naming architecture separates the task of the naming service into two distinct functions: *naming* an object and *locating* an object (see Figure 1).

Our naming service binds one or more (human readable) names to an **object handle**. An object handle is used to designate an object, and is location independent. It names the object throughout the

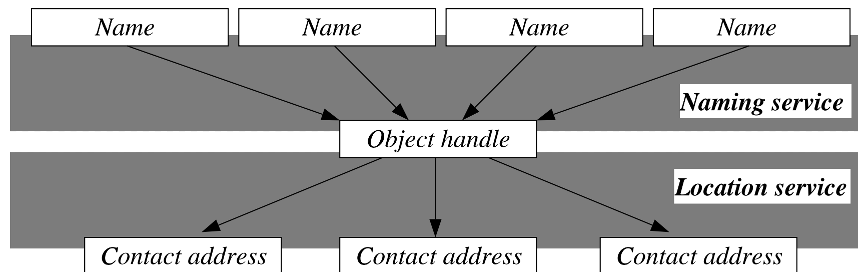

Figure 1: Two level naming scheme

object's entire life time. It does not change when the designated object moves to a different location or when it is replicated. In our approach, a traditional naming service can be used to map object names to object handles, since the binding between the names and handles is relatively stable.

In contrast, our location service provides a mapping from an object handle to one or more contact addresses. This mapping is allowed to change frequently. The location service should scale well with respect to the number of objects supported and the geographical area serviced. 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 network delays.

The location service provides three basic operations: inserting, deleting, and looking up contact addresses. The insert and delete operations are referred to as update operations. The look-up operation searches for contact addresses of a designated object. An overview of our location service can be found in [9].

2.2 Location service structure

To efficiently update and look-up contact addresses, we organize the underlying wide-area network as a hierarchy of geographical, topological, or administrative **domains**, similar to the organization of DNS. For example, a lowest level domain may represent a campus-wide network of a university, whereas the next higher level domain represents the city where that campus is located. The lowest level domains are called **leaf domains**. Each domain is represented in the location service by a **directory node**. The directory nodes together 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 considerable more efficient look-ups, as we explain in [9]. Algorithmic details of our location service can be found in [10].

Figure 2 shows an example of a tree for one specific object. In this example, leaf node N4 stores

contact addresses. A path of forwarding pointers therefore exists from root node N0, via N1 and N3, to leaf node N4. Node N1 stores, besides the forwarding pointer, addresses Addr-1 and Addr-2 from the domain of its child node N2.

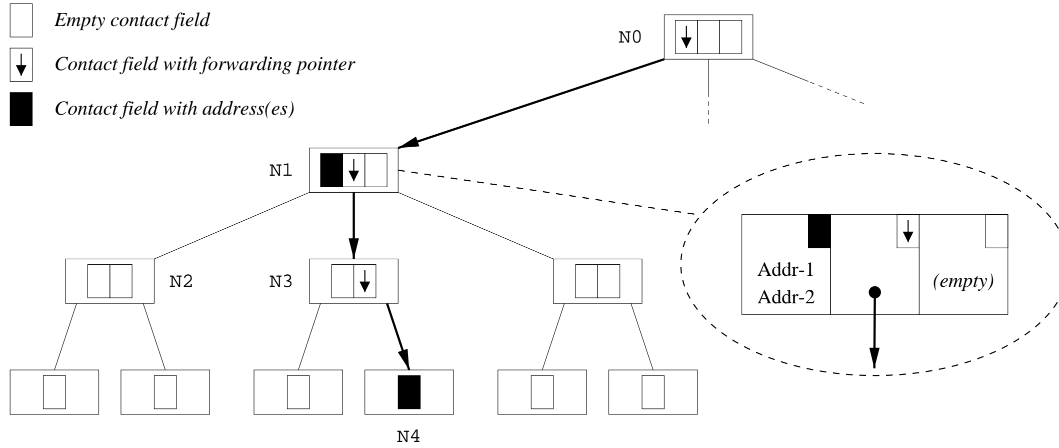

Figure 2: The organization of contact records in the tree for a specific object

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

Communication between nodes is based on Remote Procedure Calls (RPC) [11]. Using an RPC mechanism allows us to implement the update algorithms at a high level of abstraction, shielding the update algorithms from communication and scheduling issues. For update operations each node communicates with only its parent. The execution of an RPC usually involves performing another RPC at the parent, leading to cascaded invocations. In the case of update operations it 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, leading to inconsistencies between nodes. The main 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. The only problem for look-up operations we are therefore faced with is to ensure progress 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

The insert operation is illustrated in Figure 3. It consists of an upward phase in which the proper level to store the contact address is decided (Figure 3a), and a downward phase in which the path of forwarding pointers is created and the contact address is inserted (Figure 3b). 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.

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 using its parent's reply and its own preliminary decision what to do itself. 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.

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**.

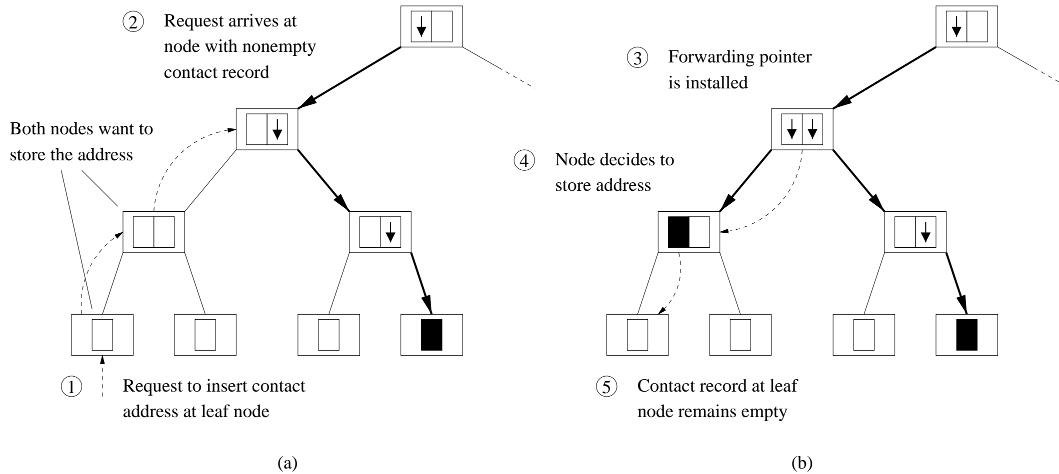

Figure 3: Insert operation

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 obtaining the lock. 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. In this figure there are two concurrent update operations, a delete and an insert. Neither node N3 nor N1 wants to store contact addresses. However, if node N1 executes the insert operation first, the node is forced to store the contact address by consistency rule 1. If node N1 executes the delete first, the insert will propagate to the root, which will subsequently store the contact address.

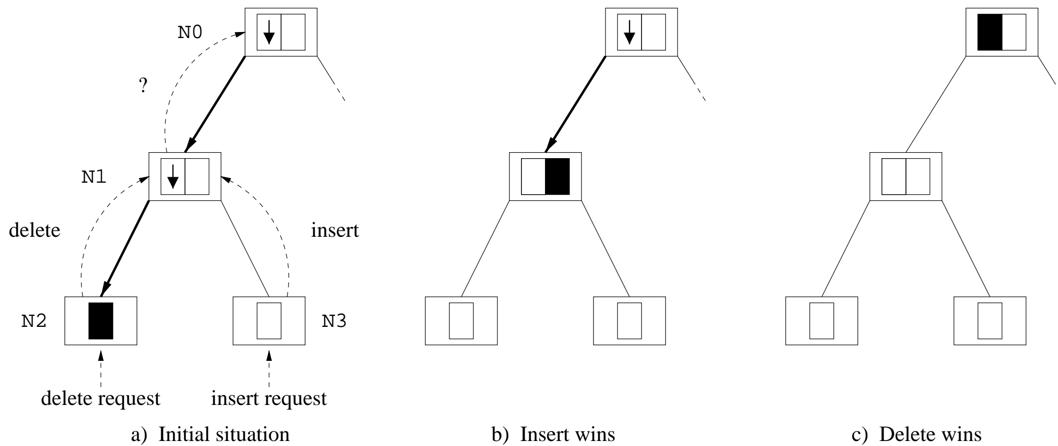

Figure 4: Race condition between a delete and an insert operation.

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 [12]. That is, the 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 that no media failures occur. Our model assumes atomic disk writes and disks that are not affected by a node crash [13]. A node crash therefore results only in the permanent loss of main memory, and thus all tentative modifications.

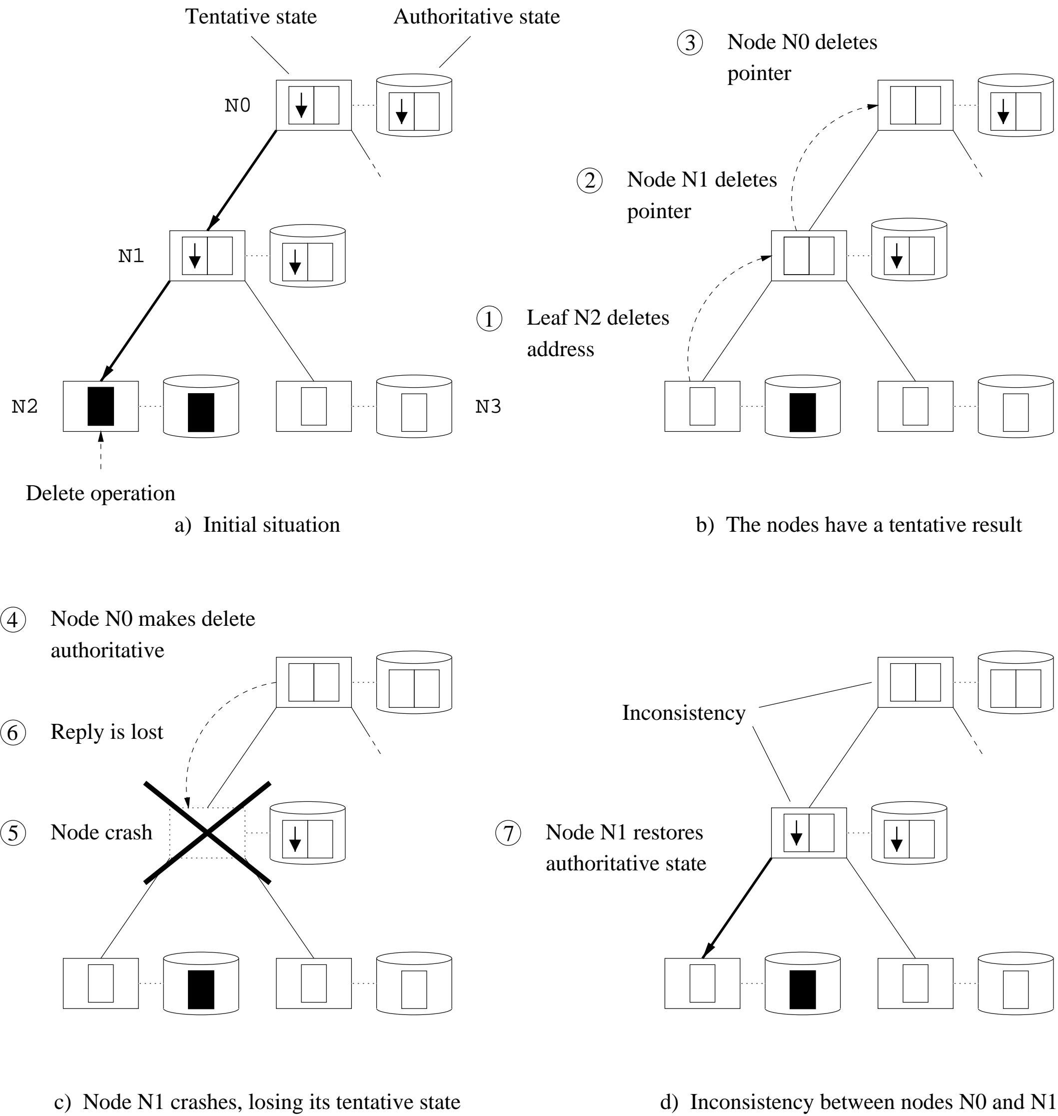

Figure 5: Occurrence of inconsistency.

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 the children retransmit their outstanding requests and the recovering node perform the requested operations, possibly again. No separate correction phase is used.

After the crash, the recovering node enters the **recovery phase**, and informs its children it is recovering from a crash. Upon receiving the news that the parent is recovering, each child retransmits its outstanding requests in the 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 the by the parent, the parent will find the associated contact field empty. In that case, it 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.

Unfortunately, during the recovery phase the operations have to deal with the inconsistencies between the two contact records. These inconsistencies mainly threaten the correct execution of the insert operation. Normally, a node decides to inform its parent of an insertion using information from its own contact record. If the contact record already contains contact addresses or forwarding pointers, the parent the parent is assumed to have a forwarding pointer, and the node does not contact its parent. This decision assumes that the contact record is consistent with that of its parent, which is the case in the absence of node crashes. 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. (The delete operation always makes the correct decision on informing the parent.)

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 no problem given

the idempotent nature of inserting a forwarding pointer: only one forwarding pointer is stored.

4.4 Recovery phase

The recovery phase lasts until all contact records at the recovering node are consistent with those at the parent. The problem is how to determine efficiently when this point is reached. Every inconsistency is caused by an update operation that had partly completed at the crashed node, and for which the crashed node was awaiting an answer from the parent. That same inconsistency will disappear if the node simply re-executes the update request as if nothing had happened. A node thus becomes consistent when it has executed all operations that were broken off by the node crash. Therefore, when each child of the recovering node has finished sending *previously issued* update requests, the recovery phase is finished.

The recovery phase is implemented by distinguishing the RPC requests *resent* after a node crash as **recovery requests**. A special **end-recovery-phase** message signals the end of the 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.

4.5 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 written to disk with one 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 as well. Being idempotent allows the modifications to be redone without adverse effect. 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.

4.6 Multiple node crashes

The recovery mechanism is *transparent* to the update algorithms at the parent and children of the recovering node. The update algorithms at a child node are unaware of the RPC system sending repeated invocation requests. The update algorithm requests an RPC only once. When its result returns, the operation is guaranteed to be performed at the parent, possibly more than once. The

update algorithms at the parent node are unaware of receiving the same RPC request message multiple times, they deal with them as separate requests. This does not lead to problems because the update operations are idempotent.

It follows from this recovery transparency at child and parent, that multiple concurrent node crashes can be handled without extra effort. When two nodes not on the same path from leaf to root crash, their recovery is unrelated. When two nodes crash on the same path from leaf to root but which are not directly linked in the tree, no problem arises as well. In both case the recovering nodes have normal running child nodes which retransmit their lost requests.

The question is what happens if two directly linked nodes (parent and child) crash. In this case, the recovering child will receive retransmissions from the grandchildren. During its recovery the recovering child will transmit its requests to the recovering parent, just as in normal recovery of the parent. The algorithms at parent and child are completely unaware of the other's recovery. The recovering child node will send its end-recovery-phase message, when it has recovered.

4.7 Leaf nodes

A leaf node does not have any children to retransmit the lost requests. Every leaf node therefore has a *persistent* message log to store incoming requests from clients of the location service. The message log is part of the RPC mechanism. The RPC mechanism saves every incoming request message to the log before handling it. After the operation is completed and a reply message is sent back to the client, the request is deleted from the log. A possible way to minimize the overhead of logging is the use of non-volatile RAM as persistent storage. A leaf node replays its message log during the recovery phase after a crash.

5 Prototype implementation

To test our ideas, we have built a prototype of our location service. In this prototype, each node has three layers: the algorithm layer, the RPC layer, and the Messenger layer (see Figure 6).

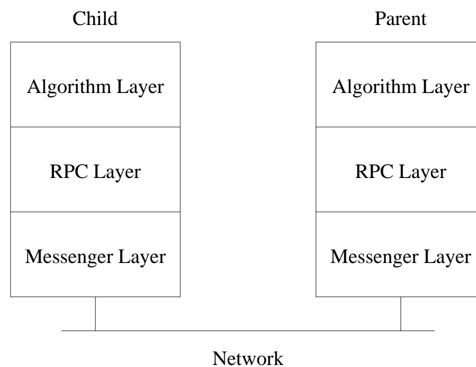

Figure 6: Layers in a node

The algorithm layer implements of the insert, delete, and look-up operation. It deals with accessing and modifying contact records from the node's contact record database. This layer is unaware of the scheduling and communication done by the lower layers, or the crash recovery performed by other nodes.

The RPC layer contains the implementation of the RPC mechanism and scheduling policy. It will send a request message and wait for a reply message. It also starts operations upon the receipt of request messages. The scheduling policy deals with the locking of contact records. The RPC layer implements at-least-once semantics [14]. The RPC layer records all outstanding requests, and retransmits the request messages when they are lost due to a node crash. Only request messages are retransmitted; reply messages can be lost when the peer node crashes. The RPC replies from orphaned operations are ignored by the node.

The Messenger layer implements reliable ordered message passing between nodes. It ensures that a message is delivered exactly once by the recipient. This reliability is guaranteed only when both nodes do not crash. It also ensures that messages will be delivered in the order they were sent. The Messenger layer informs the RPC layer when a peer node has restarted after a crash. The Messenger layer use UDP as transport mechanism.

6 Comparison

Crash recovery in the location service is a simplified form of sender-based message logging [15]. We can use a simplified form, since we have full knowledge and control over the application level. An important difference is message ordering. Sender-based message logging uses a three-way message exchange to log the receive order of messages from different hosts. This ensures that messages from different hosts can be repeated in exactly the same order. Since the update algorithms do not require the same ordering of requests between children when retransmitting, this part of the protocol is not needed. We log only request messages: we allow reply messages to be lost when a node crashes, since the update algorithms are idempotent. The node makes a checkpoint by saving the changes to disk before it sends the reply message and finishes the local operation. We use a hybrid approach to deal with leaf node failures. To avoid burdening the location service client with message logging, leaf nodes log *incoming* messages to disk themselves.

Message logging systems [16] have to deal with the problem of orphan processes. They are either avoided by making the logging procedure synchronous (in pessimistic systems), or the orphan processes are killed [14] and their changes undone (in optimistic systems). Our location service does neither. Orphan process are allowed to continue, since their changes are never incorrect. When an update operation is restarted, it will find its changes are already done by the orphaned process. This is no problem, since the update operations are idempotent.

The Rover toolkit [17] simplifies the programming of mobile aware applications. Mobility introduces the need to deal with intermittent connections. If the mobile application needs to communicate with a server application but no connection exists, the mobile application stores its RPCs in a log. These so-called Queued RPCs are transmitted when the connection is restored. Meanwhile, the mobile application continues as though the RPCs have succeeded. Our location service uses the same method to deal with unavailable hosts, although the log is not stored on disk. Queued RPC uses an asynchronous programming model in which a handler, associated with an RPC, is called when the RPC is actually performed. The location service uses a synchronous model in which the thread performing the RPC is suspended, its modification is however already made visible.

The DNS name service [7] does not have any problem with update operations, since it basically deals only with look-up operations. Data is changed directly in the name server database and no changes need to be propagated. DNS uses replication, but allows inconsistencies to exist temporarily. The replicas follow the primary node where the data is changed only in a lazy fashion.

7 Conclusion

In our location service, the use of sender-based message logging provides an easy way to implement crash recovery. Based on inherent properties of the system, such as idempotent and commutative set-like operations, the crash recovery basically comes for free. No special correction phase is needed. The same operations that have to be executed anyway, are used for recovery. This in contrast to a more heavy-weight general crash recovery mechanisms, which attempt to restore the lost state.

Future work will consist of performing further measurements on our prototype. Another important area is to look at other ways to determine the end of the recovery phase. The current implementation requires that parent and child communicate regularly to determine whether the peer node is still alive. Designing an efficient disk subsystem that implements the assumptions used by the crash recovery algorithm, will also require further research.

References

[1] M. Shapiro, P. Dickman, and D. Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.

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

[3] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile Agents: Are They a Good Idea. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, Mar. 1995.

[4] ObjectSpace, Inc. Voyager User Guide. http://www.objectspace.com/, Dec. 1997.

[5] C. Perkins. IP Mobility Support. RFC 2002, Oct. 1996.

[6] M. van Steen, P. Homburg, and A. S. Tanenbaum. The Architectural Design of Globe: A Wide-Area Distributed System. Accepted for publication in IEEE Concurrency, 1999.

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

[8] S. Radicati. *X.500 Directory Service: Technology and Deployment.* International Thomson Computer Press, London, 1994.

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

[10] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic Design of the Globe Wide-Area Location Service. Accepted for publication in The Computer Journal, 1998.

[11] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, Feb. 1984.

[12] 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*, pages 3–24. Springer-Verlag, Berlin, 1995.

[13] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, XEROX Palo Alto Research Center, Apr. 1979.

[14] F. Panzieri and S. Shrivastava. Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing. *IEEE Trans. on Software Engineering*, 14(1):30–37, Jan. 1988.

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

[16] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Trans. on Software Engineering*, 24(2):149–159, Feb. 1998.

[17] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Trans. on Computers*, 46(3):337–352, Mar. 1997.