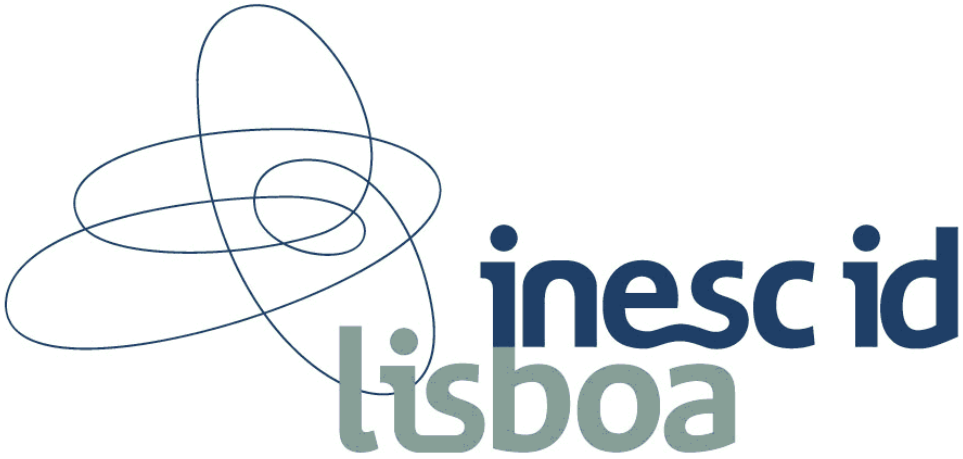

Technical Report RT/015/03

Information sharing in mobile networks: a survey on replication strategies

João Pedro Barreto
Instituto Superior Técnico/Distributed Systems Group, Inesc-ID Lisboa
joao.barreto@inesc-id.pt

September 2003

Abstract

The emergence of more powerful and resourceful mobile devices, as well as new wireless communication technologies, is turning the concept of ad-hoc networking into a viable and promising possibility for ubiquitous information sharing. In such context, replication is a key issue in order to achieve acceptable performance and availability levels.

However, the inherent characteristics of ad-hoc networks bring up new challenges for which most conventional replication systems don't provide an appropriate response. Namely, the lack of a pre-existing infrastructure, the high topological dynamism of these networks, the relatively low bandwidth of wireless links, as well as the limited storage and energy resources of mobile devices are issues that strongly affect the efficiency of any distributed system intended to provide ubiquitous information sharing.

Such aspects demand solutions that are able to offer high availability, in spite of the expected frequent network partitions and device suspension periods. Pessimistic approaches for replication are usually too restrictive solutions to achieve such requirement. On the other hand, optimistic replication strategies offer weak consistency guarantees which may not reflect the expectations of users and applications.

This paper aims at exposing the main challenges of a replicated system operating in the particular environment of mobile ad-hoc networks, with the goal of providing general guidelines for an effective solution. Based on a common conceptual model, some key design strategies are described and compared. A survey of some relevant state of the art solutions illustrates such concepts and presents implementations of the main design alternatives. In conclusion, guidelines are proposed towards an effective replication strategy for mobile ad-hoc networks.

Contents

1 Introduction

The evolution of the computational power and memory capacity of mobile devices, combined with their increasing portability, is creating computers that are more and more suited to support the concept of ubiquitous computation [Wei91]. At the same time, novel wireless communication technologies have provided these portable devices with the ability to easily interact with other devices through wireless network links.

As a result, users are progressively using mobile devices, such as handheld or palmtop PCs, not only to perform many of the tasks that, in the past, required a desktop PC, but also to support innovative ways of working that are now enabled.

Some existing systems already allow users to access their personal documents using their mobile devices while they are away from the desktop PCs where the documents are typically stored.

Evolving from this initial concept of isolated disconnected operation, more interesting scenarios can be envisioned if one also considers the possibility of direct interactions between the disconnected mobile devices. Instead of disconnected users working in isolation over their personal data, such scenarios would enable actual collaboration to occur between the interacting mobile peers.

Many real life situations already suggest that their users could benefit substantially if allowed to cooperatively interact using their mobile devices and without the requirement of a pre-existing infrastructure. A face-to-face work meeting is an example of such a scenario. The meeting participants usually co-exist within a limited space, possibly for a short period of time and may not have access to any pre-existing fixed infrastructure. Under such co-present collaborative activities [LH98], participants hold, manipulate and exchange documents that are relevant to the purposes of the meeting.

If each participant holds a mobile device with wireless capabilities, a spontaneously formed wireless network can serve the purposes of the meeting. These wireless networks, possibly short-lived and formed just for the needs of the moment, without any assistance from a pre-existing infrastructure, are normally referred to as ad-hoc networks [Shr02].

One key challenge in supporting such collaborative scenarios consists of providing effective means for information sharing between the mobile users. Replication mechanisms are usually employed for the purpose of achieving acceptable performance and availability levels [RRPK01]. However, the nature of the scenarios we are addressing entails significant challenges to a replication solution to be devised.

The high topological dynamism of mobile ad-hoc networks entails frequent network partitions. On the other hand, the possible absence of a fixed infrastructure means that most situations will require the services within the network to be offered by mobile devices themselves. Such devices are typically severely energy constrained. As a result, the services they offer are susceptible of frequent suspension periods in order to save battery life of the server's device. From the client's viewpoint, such occurrences are similar to server failures.

These aspects call for solutions which offer high availability, in spite of the expectedly frequent network partitions and device suspension periods. Pessimistic approaches for replication are usually too restrictive solutions to fulfill such a requirement. On the other hand, optimistic replication strategies offer weak consistency guarantees which may not reflect the expectations of users and applications.

Whichever strategy is taken, it must also take into account the important limitations in memory resources and processing power of typical mobile devices, as well as the reduced bandwidth of wireless links, when compared to other wired technologies.

Additionally, the novel usage scenarios that are commonly regarded in co-present collaborative activities suggest that the set of operations that conventional systems offer to users and applications is insufficient. A richer set of operations should be available in order to more closely reflect the way we are used to performing co-present collaborative activities without the assistance of a mobile ad-hoc network.

This paper presents a survey of existing state of the art solutions which constitute relevant approaches to the problem of supporting information sharing in mobile ad-hoc networks through replication. Notice that some of the solutions presented in the next sections were not initially designed with the intent of supporting the mobile ad-hoc environments. Instead, their reference in the survey is justified by the relevance of some specific aspects of

their design in relation to the subject of this paper.

The remaining of the paper is as follows. Section 2 introduces a common system model which is referenced throughout the paper. Section 3 discusses the trade-off between correctness and availability. Sections 4 and 5 present the main design aspects and alternatives of pessimistic and optimistic replication strategies, respectively. Sections 6 to 11 are dedicated to describe existing implementations, namely: Coda, Gossip Framework, Bayou, Roam, Deno and AdHocFS. Section 12 concludes such analysis by comparing the main characteristics of each surveyed system. Finally, Section 13 draws the main conclusions of the present paper and proposes guidelines for a replication strategy for mobile ad-hoc networks.

2 System Model and Terminology

In order to present a comprehensive comparative analysis between the solutions that are surveyed in this paper, this section presents a common system model for reference throughout the remaining sections.

Data in a replicated system consist of a set of logical objects. A logical object can, for instance, be a database item, a file or a Java object. Such a logical object is implemented by a collection of physical copies, called replicas, each stored at a single location within the system.

Each replica is held by a *replica manager*, which contains the replica and performs operations upon it. Such operations can be reduced to the basic *read* and *update* operations. Moreover, we assume for simplicity that each replica manager maintains a replica of every object. Ratner *et al.* [RRPG99] propose a solution for allowing replica manager to replicate only a subset of the logical objects. For the sake of generality, the set of replica managers may be dynamic, and thus change with the creation or removal of new elements.

Hereafter, we assume an asynchronous system in which replica managers can only have fail-silent faults [Tan95]. Network partitions may also occur, thus restricting connectivity between replica managers which happen to be located in distinct partitions.

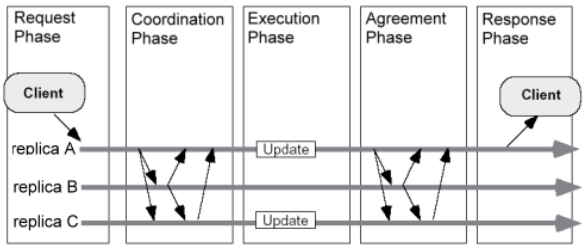

Figure 1: Phases involved in an replica operation request.

When a replica manager receives an update request, it is assumed that the update is applied recoverably upon the replica. Consequently, an individual replica manager will not reach an inconsistent replica value if it crashes during the execution of the operation. Moreover, we assume that replica managers apply update operations atomically to their replicas, so that its execution is equivalent to performing those operations in some strict sequence.

An operation request sent to a replica manager will generally involve five distinct phases [WPS+00]. Notice that the actual actions performed in each phase are specific to each particular replication solution. Moreover, some solutions may skip some phases, order them in a different manner or iterate over some of them. The phases are as follows:

Request. The client submits an operation upon a logical object to one (or possibly more) replica managers.

Coordination. The replica managers coordinate in order to perform the operation request consistently. Namely, a decision is made on whether the operation is to be applied and on the ordering of this request relative to others.

An update request that has successfully completed its coordination phase becomes a *stable update* request. In contrast, an update request whose coordination phase hasn't yet been completed is referred to as a *tentative update*.

Execution. The replica managers execute the request upon their replicas.

If the operation request to be executed is a tentative update, it is executed tentatively. In this case, the replica manager must ensure that the effects of the tentative update can be later undone, depending on the coordination decision.

Agreement. The replica managers reach a consensus upon the effect of the requested operation.

Response. One or more replica managers responds to the client that issued the request. In case of a read operation, the result consists of the value obtained after performing the query. In case of an update operation, the result indicates whether it was successful or not.

Replication strategies can be distinguished between pessimistic and optimistic [DGMS85]. A fundamental difference between such replication paradigms is concerned with the order of the five phases presented above. Section 4 and Section 5 will address each of these paradigms in greater detail.

Finally, it should be emphasized that it is not the intention of this survey to address the security issues that may concern a replicated system in mobile ad-hoc environments. Therefore, we assume a naive trust model which does not consider the existence of any distrussed entity. More realistic trust models for replicated systems can be found in [RPC+93], [KBC+00], [RD01] or [BI03].

3 Correctness versus availability

When one considers centralized access to data objects of any type, an implicit notion of correctness is usually well known and accepted by users and applications. For instance, database systems generally offer the properties of atomicity and serializability as essential components of their correctness guarantees [DGMS85]. Moving to a different context, most file systems supply familiar file sharing semantics such as Unix semantics [LS90].

In a replicated system, however, a logical object may be replicated in various physical replicas within the network. In this case, ensuring that each physical replica is individually correct according to the above properties is not sufficient. Instead, the consistency amongst all physical replicas of a logical object must also be considered.

Consider, for instance, that a logical object containing information concerning a bank account is replicated in two physical replicas, located at distinct machines. Initially, the bank account had a balance of 100 in both replicas. Now assume that the physical replicas become separated by a network partition. If both of these replicas are allowed to be updated while the network is partitioned, an incorrect replica state may be reached. For example, if a withdraw request of 60 is performed at one of the partitions and then a withdrawal of 70 is also accepted at the other partition. Since the replicas are unable to contact each other, the updates issued at each one are not propagated between them. Hence, an inconsistent state is reached, since the logical bank account can be seen as having different balances depending on which replica the read request was made on. Moreover, the bank account semantics were violated, since a total withdrawal of 130 was, in fact, allowed when the account had insufficient funds for it.

One possible solution for ensuring that the bank account semantics are verified in the previous example would be to prohibit any update or read request to the physical replicas in the presence of network partitions. Conversely, the same behavior could be followed in the presence of a replica's machine failure. This would be a pessimistic strategy, which prevented inconsistencies between replicas by restraining their availability. As a result, strong consistency guarantees would be ensured, which would fulfil the correctness criteria of most applications.

When network partitioning or server failure are not negligible events, availability and correctness become two competing goals that a replicated system must try to achieve to some extent [YV]. The trade-off between these contending vectors can yield replicated solutions which offer weaker correctness guarantees as a cost for improved availability. Those solutions are designated as optimistic strategies.

One important aspect to stress out is that different applications can have different correctness criteria regarding the replicated logical objects they access. Therefore, weaker consistency guarantees provided by optimistic strategies may still meet the

correctness criteria of some application semantics.

For instance, some applications may consider it acceptable for replicas to temporarily have distinct values, provided that they eventually reach a correct and consistent value. In particular, if a bank account was allowed to have a negative balance, the above example might be acceptable according to such correctness criteria. In spite of having distinct balance values during the partition, the physical replicas would propagate each one's update upon being reconnected. In the end, both physical replicas would have the same consistent balance value, -130.

The next subsections present some consistency guarantees that are normally offered by replicated systems. All of them are described with reference to a general situation where each client i performs operations upon a logical object, deriving an execution $o_{i0}, o_{i1}, o_{i2}, ...$ Each o_{ij} represents an operation requested by client i. It is assumed that operations are synchronous, that is, client i waits for the completion of operation o_{ij} before requesting o_{ij+1}.

3.1 Strong consistency guarantees

In a replicated system, one can consider a virtual interleaving of the clients' operations which reflects one possible correct execution if a centralized system was used. The strong consistency guarantees which are described ahead take one such possible interleaving as a correctness reference to the actual interleavings which are performed at each physical replica.

Enforcing strong consistency guarantees requires adopting pessimistic strategies for replication, which will be presented in Section 4.

3.1.1 Linearizability

The most strict consistency guarantee is *linearizability* [GCK01]. A replicated object is said to be linearizable if, for any series of operations performed upon it by any client, there is some canonical interleaving of those operations such that:

(1) The interleaved sequence of operations meets the specification of a single correct copy of the objects.

(2) The order of operations in the canonical interleaving is consistent with the *real times* at which the operations occurred in the actual execution.

The definition of linearizability makes use of a virtual canonical execution which interleaves all the operations requested by clients in such a way that, if applied to a virtual single image of the object would, produce a correct object value. In order to be linearizable, each physical replica of the object must be, at any moment, consistent with the real times at which the operations were applied in the canonical interleaving. This means that linearizability captures the idea that each physical replica should reflect the actual times at which the canonically interleaved operations happened. Though desirable in an optimal replicated system, such real-time requirement is normally impractical under most circumstances [GCK01].

3.1.2 Sequential consistency

A less strict consistency criterium is called *sequential consistency*. Basically, it discards the real-time requirement of linearizability and only looks at the order of operation requests performed at each client, the *program order*. More specifically, the second condition of the linearizability criterium is now changed to:

(2) The order of operations in the canonical interleaving is consistent with the program order in which each individual client executed them.

Notice that sequential consistency does not imply that every operation interleaving at each replica has the same total order. Instead, different interleavings can be applied at distinct replicas, provided that each client's operation order is preserved at each of these different interleavings. Such interleavings must also guarantee that the result of each operation is consistent, in terms of the objects' specification, with the operations that preceded it in the canonical interleaving.

The definition of sequential consistency is similar to a transactional property designated as *one-copy serializability*. However, sequential consistency, just as every replication consistency guarantee referred in this paper, does not include any

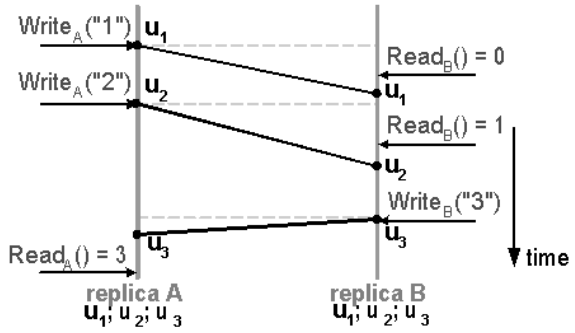

Figure 2: An example of an execution that is sequentially consistent, with respect to the canonical interleaving: $Read_B() = 0$, $Write_A("1")$, $Read_B() = 1$, $Write_A("2")$, $Write_B("3")$, $Read_A() = 3$. Notice that such execution is not linearizable, since the values returned by both $Read_B$ operations are not consistent with the real times at which both $Write_A$ operations occurred.

notion of operation aggregation into transactions. For this reason, sequential consistency and one-copy serializability should be regarded as distinct concepts.

3.2 Relaxed consistency guarantees

Relaxed consistency guarantees are normally specified in terms of the properties that are ensured in the way updates are ordered at each replica. In contrast to strong consistency guarantees, the notion of a canonical operation interleaving is no longer used as a reference by relaxed consistency guarantees.

The inherently weak requirements of such guarantees enable optimistic replication strategies to be employed. Section 5 describes such strategies.

3.2.1 FIFO ordering of updates

One weak consistency guarantee is *FIFO* ordering of updates. It considers the partial ordering between the operations requested by each individual client. It guarantees that such partial orderings are preserved by every operation interleavings applied to the physical replicas.

This consistency guarantee might seem resemblant to strict sequential consistency. However, one important condition of sequential consistency is absent from FIFO ordering: there is no requirement

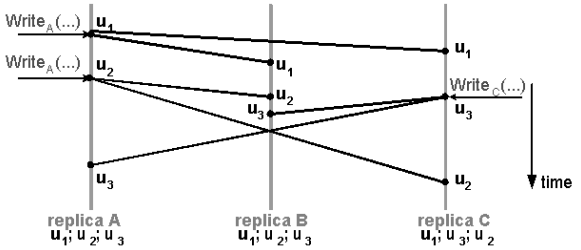

Figure 3: An example of FIFO ordering of updates: the partial order between operations requested by each individual client is ensured at every replica. Notice that, since FIFO ordering is not total, not every replicas apply the updates in the same sequence.

for each operation to be consistent with the operations that preceded it at some canonical interleaving.

3.2.2 Causal ordering of updates

A stronger consistency guarantee is that of *causal* ordering of updates. This guarantee is based on the *happened-before relationship* between events, $\rightarrow$, as defined by Lamport [Lam78]. In the particular case of updates issued in replicated system, this relationship relies on the following points:

(1) If two updates, u_1 and u_2, were issued at the same replica manager i, then $u_1 \rightarrow u_2$.

(2) Whenever an update is propagated between two replica managers, the event of sending the update occurred before the event of receiving the update.

(3) If e, e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

A system guarantees causal ordering of updates if and only if the partial causal ordering between operations is verified at every operation interleaving applied to each replica.

Since all operation requests issued by the same client are causally related, any system providing causal ordering of updates also provides FIFO ordering.

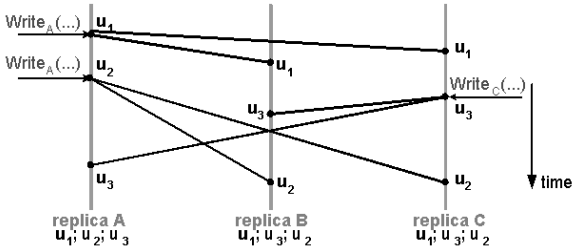

Figure 4: An example of causal ordering of updates: the happened-before relationships between updates ($u_1 \rightarrow u_2$ and $u_1 \rightarrow u_3$) is ensured at the update ordering at every replica. Notice that, since causal ordering is not total, not every replicas apply the updates in the same sequence.

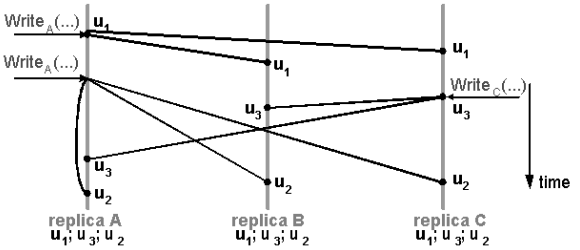

Figure 5: An example of total ordering of updates: the total order $u_1 < u_3 < u_2$ is ensured at every replica.

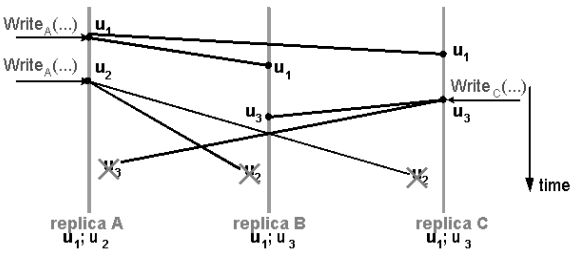

Figure 6: An example of strict causal ordering of updates: to ensure strict causal ordering, replicas have to discard certain concurrent updates (u_2 and u_3 are concurrent).

3.2.3 Total ordering of updates

One other weak consistency guarantee that can be considered is *total* ordering of updates. A system offers total ordering if the operation interleaving at every replica respects a total ordering established between all operations issued in the system.

The linearizability consistency criterium guarantees total ordering of updates by ordering operations by the real times at which they were performed. Note that, however, any more practical total ordering between operations can be employed.

One important property of a system offering total ordering of updates is that is guarantees that, if any set of replicas receives the same operation requests, their physical values will be identical. This is not true with the other weak consistency guarantees described previously. With FIFO ordering, operations issued by different clients can be differently ordered at distinct replicas, thus resulting into possible different replica values. The same can happen with causal ordering when operations are concurrent with respect to the happened-before relationship.

Finally, weak consistency criteria can be combined in order to provide stronger consistency guarantees. One common consistency criterium is obtained by ensuring both total and causal ordering of updates.

3.2.4 Strict causal ordering of updates

Some systems require a stronger variant of causal ordering of updates which eliminates the possibility of causally concurrent updates. Strict causal ordering offers such guarantee.

A replicated system offers strict causal ordering of updates if and only if: given an update u_i, for every update u_k preceding u_i at some update interleaving applied at some replica, the following condition met: $u_k \leq u_i$.

Contrary to the other weak consistency guarantees described so far, concurrent updates cannot be accommodated by one replica exhibiting strict causal ordering of updates. Instead, in the presence of multiple concurrent updates, a replica must accept only one of them and discard the others.

An important property of replicated systems that provide strict causal ordering of updates if that each operation is only applied in the context in which it was issued. That is, if an update operation was issued upon a replica when its value was a result of a specific set of older updates, the new update, if accepted by the replica, will be ordered immediately after those older updates.

3.3 Hybrid solutions and eventual sequential consistency

Some solutions offer two levels of consistency guarantees to applications, combining both pessimistic and optimistic replication strategies.

These solutions are based on optimistic replication strategies. As a result, replicas offer weak consistency guarantees as a trade-off for high availability. The update operations and the replica values that result from this optimistic operation are normally designated as *tentative*.

On top of the optimistic strategy, a pessimistic replication scheme is applied to ensure stronger consistency guarantees. The objective of the pessimistic scheme consists in restricting the set of *tentative* updates that are applied to a replica's value. The updates that constitute this stricter collection of updates are typically designated as *stable* or *committed* updates. Conversely, the value that results from the application of the stable updates is called the *stable* or *committed* value.

One particular case of a hybrid solution is that where the replica's stable value is sequentially consistent, according to the definition introduced in Section 3.1.2. It should be noted, however, that such sequential consistency guarantee applies only to the set of stable updates.

From the viewpoint of the clients that issue tentative updates, the only guarantee is that such tentative updates will eventually be placed in the canonical interleaving that defines the sequentially consistent stable value. Such event corresponds to completing the coordination phase of each tentative update, when it becomes a stable update. In the meantime, the stable value will simply not reflect the not tentative updates which have not become stable. Hereafter, we shall refer to this consistency guarantee as *eventual sequential consistency*.

The advantage of this approach is that it can easily accommodate different applications with distinct correctness criteria and, consequently, distinct consistency requirements. Applications with stronger correctness criteria can select to access the stable view of replicated objects. Applications with less demanding correctness criteria can enjoy the higher availability of the tentative view.

4 Pessimistic strategies

Pessimistic strategies prevent inconsistencies between replicas by restraining the availability of the replicated system.

Each replica manager makes worst-case assumptions about the state of the remaining replicas. Therefore its operation follows the premise that, if any inconsistency can occur in result of some replica operation, that operation will not be performed. As a result, pessimistic strategies yield strong consistency guarantees (Section 3.1).

A pessimistic replication protocol performs the five phases in the canonical order described in Section 2 (request, coordination, execution, agreement and response), though some particular solutions may skip the coordination and/or agreement phases. This means that, after issuing an operation request, a client has to wait for all the remaining phases to complete before obtaining a response. If a network partition or a failure of some replica manager prevents the coordination or agreement phases from performing their distributed algorithms, then the request response will as well be disrupted. As a consequence, the replicated system's availability is reduced.

The following subsections present three representative pessimistic replication strategies which are relevant for some solutions discussed in the rest of the paper. For a more exhaustive analysis, the reader should consult [DGMS85] and [WPS+00].

4.1 Primary copy

This approach [AD76, Sto79] assumes the existence of a single distinguished replica, designated as the primary copy. Every update operation to the logic object must be handled by the replica manager holding its primary copy. Updates are then propagated to the remaining replicas.

In the case of reads, a lock has to be acquired at the primary copy's replica manager. The actual read operations can then be performed at any replica of the logical object.

In the event of a network partition, only the partition containing the primary copy is able to access it. Upon recovery, updates are forwarded to the previously partitioned sites to regain consistency.

Under situations where network partition is distinguishable from the failure of a node, a new pri-

mary copy can be elected when the previous one fails. However, if it is not possible to determine if whether the primary copy is unavailable for failure or partition reasons, the system must always assume that a partition occurred and no election can be made.

4.2 Tokens

This approach [MW82] bears close resemblance to the primary copy scheme. The exception is that the primary copy of a logical object can change for reasons other than network partition. Each logical object has an associated token, which allows the replica holding it to access the object's replicated data.

When a network partition takes place, only the partition which includes the token holder will thus be able to access the corresponding logical object. One disadvantage of this approach lies in the fact that the token can be lost as a result of a communication or replica manager failure.

One variation of this basic token protocol is the *single writer multiple readers* [LH86], where two types of tokens are obtainable: read and write tokens. Several replica managers can simultaneously hold a read token on a particular logical object, which enables them to serve read requests to the local replicas of that object. On the other hand, a replica manager can hold a write token, which implies that no other replica manager holds a token of any type on the same logical object. With a write token, a replica manager can serve both update and read requests to its local replica of the object.

4.3 Voting

In a voting strategy [Gif79], every replica is assigned some number of votes. Every operation upon a logical object must collect a read quorum of r votes to read from a replica or a write quorum of w votes to update a replica value.

The following conditions must be verified by the r and w quorum values:

(1) $r + w$ exceeds the total number of votes, v assigned to a logical object, and

(2) $w > v/2$

The first condition ensures that the intersection between read and write quora is never null. In a partitioned system, this means that a logical object cannot be read in one partition with read quorum and written in another partition with write quorum. This would lead to inconsistencies, since updated replicas in the write quorum partition would not be reflected in the values read in the read quorum partition.

The second condition guarantees that only a maximum of one write quorum can be formed. This way, in the event of network partitions, the case of two or more partitions holding a write quorum for a logical object will never happen.

Notice that, if r is chosen so that $r < v/2$, it is possible for a logical object to be read by replica managers on more than one partition, in which case update operations are not allowed in any partition. High availability for read operations can thus be achieved by choosing a small r value.

One main aspect differentiates this strategy from the ones previously presented. This strategy does not have to distinguish between communication failures, replica manager failures or network partitions. Voting is simply carried out by collecting the votes of the replica managers that are available at each moment. In contrast with the primary copy and token solutions, availability is not compromised by the inability of distinguishing the cause of a replica manager's unavailability.

One drawback of the quorum scheme is that reading from a replica is a fairly expensive operation. A read quorum of copies must be contacted in this scheme, whereas access to a single replica suffices for all other schemes, provided that the read lock or token has already been acquired.

5 Optimistic strategies

In contrast to pessimistic replication strategies, optimistic strategies do not limit availability. Requests can thus be served just as long as any single replica manager's services are accessible. The result is a higher availability in comparison with the pessimistic approach.

This is achieved by ordering the *execution* and *response* phases before the *coordination* and *agreement* phases. This way, the client issuing the request does not have to wait for the replica manager to contact the other, possibly inaccessible, peers in order to complete the *coordination* and *agreement* steps. Since, from a client's viewpoint, an oper-

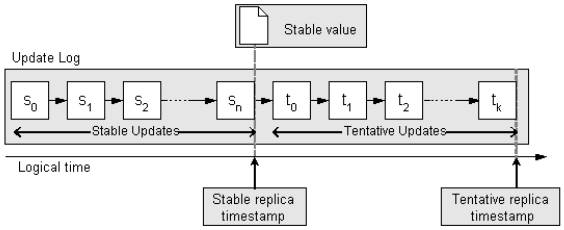

Figure 7: Generic replica state maintained by optimistic strategies.

ation request is served as soon as the client has received the response from the replica manager, a high availability is accomplished.

A consequence of anticipating the *execution* and *response* phases is that inconsistencies may occur if different replicas are updated concurrently. As a consequence, optimistic strategies typically offer weak consistency guarantees (Section 3.2).

To restore replica consistency, replica managers must detect update conflicts and, if necessary, resolve them. Hence, the *coordination* and *agreement* phases must now be responsible for dealing with detection and resolution of potential conflicts that may have occurred.

The following subsections are dedicated to some key design aspects that characterize an optimistic replication solution. Namely, replica state, version timestamping, update stabilization and log truncation.

5.1 Replica state

A replica manager maintains, for each replica it stores, a collection of data structures, illustrated in figure 7. These are:

Stable value. This is the value that is obtained after applying all the ordered stable updates that have been received by the replica manager to some initial replica value.

Update log. The update requests that have been received by the replica manager are stored in this log. Each update in the log represents a new replica version that results from applying that update after all updates ordered before it to the initial replica value. Such version is identified by a version timestamp that is assigned to the update.

The updates stored in the log can be distinguished as stable updates and tentative updates. One condition that must be verified is that a stable update is always ordered before any tentative updates in the log.

Tentative and stable replica timestamp. Figure 7 illustrates the state that is associated with each replica at some moment. Generally, a replica manager that maintains all the data structures represented in the figure is able to offer two possibly distinct views upon a replica: its stable and tentative value. The former is simply the same as stored in the stable value data structure. The latter results from the application of the tentative updates included in the log to the stable value.

Both views have an associated version timestamp. Those version timestamps correspond to the timestamps of the most recent stable and tentative updates in the log, respectively.

5.2 Version timestamping

In order to detect conflicts and determine the set of updates to be exchanged between replica managers during reconciliation sessions, a mechanism must be employed to timestamp replica versions. The following subsections present some key approaches for version timestamping, along with their main advantages and disadvantages.

5.2.1 Version Vectors

A version vector [Mat89][Fid91] is a vector of counters, one for each replica manager in the system. Each replica manager maintains version vectors as logical timestamps in order to describe the history of its own local replica. As described in the previous section, such timestamps are the tentative and stable replica timestamps, as well as the timestamps of each update in the update log.

Whenever a new version is created as a result of an update issued to a replica manager, that update is stamped with a version vector. Consider the version vector that describes the original version to

which the update was issued upon. The new version vector is simply obtained by incrementing the entry corresponding to the replica manager that generated the resulting new version.

Furthermore, when two conflicting replica versions are merged by some conflict resolution procedure, the resulting version' timestamp is calculated by the *merge* operation receiving both version's timestamps as inputs. The returned version vector has, for each entry, the maximum value of the corresponding entry in the input version vectors.

Two version vectors can be compared to assert if there exists a happened-before relationship between them. Given two version vectors, vv_1 and vv_2, vv_1 causally precedes vv_2, meaning that a happened-before relationship links vv_1 to vv_2, if and only if, the value of each entry in vv_2 is greater or equal than the corresponding entry in vv_1. If this condition is verified, vv_2 is said to *dominate* vv_1. If neither vv_1 dominates vv_2, nor vv_2 dominates vv_1, vv_1 and vv_2 are conflicting versions.

Version vectors are a simple and effective way of version timestamping. When generating and merging of replica versions, the new version vector is easily calculated by an increment or merge operation. A simple arithmetic comparison of two version vectors allows a system to conclude whether any of the associated replica versions dominates the other or, else, if a conflict exists.

The size occupied by version vectors is linearly dependent on the number of replica managers in the system. This is a significant scalability obstacle when concerning systems with a high number of replica managers. In the case of timestamps for logged updates, this problem is reduced, however, by storing only a $[replicaId, entryValue]$ pair, where $replicaId$ identifies the replica manager that received the update from a client and $entryValue$ is the entry corresponding to that replica manager if an actual version vector was stored. As long as the updates in the log are causally ordered, the timestamp of a particular logged update can be obtained by looking at the tentative replica timestamp and the $[replicaId, entryValue]$ of the updates that succeed it in the log. This approach is employed in the Bayou replicated system (Section 8).

The major limitation of this approach is that the collection of replica managers is considered to be static. Each replica manager has an pre-assigned fixed position within a version vector. This means that the creation or retirement of replica managers from the system is prohibited by this basic version vector approach. The next section presents dynamic version vectors, which address this important limitation.

5.2.2 Bayou's version vectors

An alternative solution is proposed by the Bayou replicated system [PSTT96]. This approach handles creation and retirement of replica managers as update operations. Such special updates are propagated between the remaining replica managers using the general update propagation scheme of Bayou.

Upon reception of a creation or retirement update, a replica manager proceeds with adding or removing the corresponding entry from the its version vectors. The eventual consistency guarantees of Bayou's update propagation protocol ensure that the creation or retirement update will eventually be reflected on every replica manager in the system.

An important problem arises, though, if replica managers are allowed to discard updates from their log before every other replica managers have received those updates. This is the case of Bayou.

Under such circumstances, it is possible for a replica manager, R_1, to receive a retirement update concerning replica manager R_2, process it by removing R_2's entry from its version vectors and then discard the update. During a later update propagation session with another replica manager, R_1 may be presented with version vectors referencing R_2, which it no longer knows about. In such situation, R_1 may not know about R_2 for two distinct reasons: either R_1 never hear about R_2 or R_2 was created and subsequently retired. In the former case, R_1 should add an entry for the new, previously unknown replica manager R_2 in its version vectors. In the former case, R_1 should assume that R_2 is, in fact, retired.

Bayou solves this ambiguity by using a recursive naming scheme to identify replicas managers. A new replica manager that is created by issuing a creation update upon some existing replica manager gets its identifier as a composition of the existing replica manager's identifier and the creation update's timestamp. Examining such recur-

sive identifier allows a replica manager to solve the create/delete ambiguity, which is described in detail at [PSTT96].

The recursive replica manager naming scheme, however, is a significant drawback of this approach. Since new replica manager's identifiers are constructed as a composition of an existing identifier and a version vector, membership changes to the collection of replica managers irreversibly increase the size of identifiers. If replica creation and retirement should be frequent, the benefits from removing the retired replica managers' entries from version vectors may rapidly be outmatched by the penalty of increasing replica manager identifier sizes.

5.2.3 Dynamic version vectors

Ratner *et al.* [DRP99] proposed a dynamic variation of the static version vector approach described previously. A dynamic version vector is represented by an *associative vector*, comprised of a variable number of $< ReplicaId : Counter >$ pairs. Instead of a static vector, a dynamic version vector is able to expand and compress itself in response to replica creation or deletion.

Moreover, Ratner noted that updates to a certain replica typically are performed by a small set of writers. As the elements in a version vector represent the updates issued at each replica manager, only that restricted set of writer replica managers needs to have corresponding elements in the version vector. In fact, having elements for replica managers that exclusively read from a certain replica is unnecessary, since their counters are permanently null.

From this observations, dynamic version vectors start as empty vectors and are able to expand with a new element as any replica manager generates its first update. The absence of an element for a given replica manager in a dynamic version vector is equivalent to holding a zero counter for that replica manager. In particular, the happened-before relationship described in Section 3.2.2 is still verified with absent elements instead of zero valued elements. So, replica expansion is simple: when the first update is generated at a given replica manager, it simply adds a new element in its dynamic version vector. As the update is propagated to other replica managers, so does the expanded vector, which caused those replica managers to add the new element to their dynamic version vectors.

When a replica is deleted, it should be removed from the version vector. Otherwise, the vector dimension would continually increase as replicas were created and deleted. Additionally, a replica manager that has generated some updates in the past but has ceased its update activity should also be removed to reduce the vector size.

However, vector compression is challenging because a vector element concerning a given replica can only be done once all replicas have the same value for that element. This condition ensures that the happened-before relationships amongst version vectors are equivalent to those between the same version vectors after being compressed. This condition requires that a consensus must be reached among all the replica managers to guarantee that all have the same value in the vector element to be removed.

The Roam system periodically executes a distributed algorithm to remove the elements corresponding to replica managers which have not issued updates recently. Such replica managers may have discarded the replica or have simply ceased its update activity and therefore its element should be removed from the vector.

The distributed algorithm relies on the underlying update epidemic propagation protocol of Roam. When finished, a consensus is reached on a value to be subtracted at all replica managers to the element which is being removed. If the resulting value is zero, a replica manager removes the element from its vector. During the execution of the distributed consensus algorithm, the normal operation of the system is retained, which means that any replica manager, including the one being removed, can generate and propagate updates. A more detailed description of the algorithm can be found in [Rat97].

Under circumstances where the premiss that updates are probable to occur by a small set of replica managers is verified, Ratner's dynamic version vectors are an interesting solution. Not only can it deal with the static vector's problem of replica creation and deletion, but it also optimizes vector size to reflect only the set of writers. Such aspect is of significative importance to replicated systems with a potentially high number of replicas, where scalability is a key issue.

The main disadvantage of dynamic version vectors is the overhead caused by the periodic execution of the consensus algorithm. In particular, additional network usage and memory space is required in order to perform the distributed algorithm. Furthermore, in situations where network partitions or failure of replica managers can last for long periods, consensus is equally delayed. This can severely reduce the effectiveness of vector compression. However, the characteristics of the distributed consensus algorithm guarantee that the operation of the replicated system is not affected by such situations.

5.2.4 Hash histories

The Hash History approach to version timestamping [KWK03] proposes a radically distinct strategy from that used by version vectors. Here, instead of identifying versions based on the collection of replicas that are able to issue updates upon the replicas, a version identifier is based on its actual data contents.

This is accomplished by applying a cryptographic hash function, such as SHA-1 [Nat95], to the entire contents of each version. The resulting fixed size hash value is then used as an identifier of that version. Additionally, the replication system also has to maintain (parent, child) pairs between updates in the log so that causal relationships between versions can be extracted.

This exclusively content-based approach has the advantage of avoiding the main problems associated with version vector strategies. Namely, those related to dynamic membership change of the collection of replica managers and the growth of vector size with the number of replica managers.

Furthermore, the hash history approach can take advantage of situations of *coincidental equality*. A coincidental equality occurs when two independent chains of update operations, issued at distinct replicas, produce identical version data. To a version vector approach, this event would be regarded as a conflict, since the updates that produced the replica version are causally conflicting according to the definition in Section 3.2.2. Using hash values to identify replica versions, the system would consider both versions as equivalent, thus avoiding performing unnecessary conflict resolution procedures. The probability of the event of coincidental equality is, however, strongly dependent on the update patterns that characterize the application domain of each replicated system. Consequently, for some systems, the advantages of detecting coincidental equalities may not be sufficiently relevant to account for such feature as a real advantage.

However, hash histories suffer from two main weaknesses. Firstly, a hash value has to be calculated over the entire contents of each replica version that is created. For objects with reasonable data dimensions, this represents a substantial performance overhead. Even if an update consists only of a small number of bytes, it will require sequentially accessing to the whole replica contents and performing intensive processing over it in order to obtain the hash value that identifies the new version. Using version vectors, a simple increment of one to a version element would be needed.

The second disadvantage lies in the fact that, in contrast to version vectors, hash values carry no explicit causal information. That is, given two version vectors, one can deduce, by a simple comparison of both vector's elements, whether one precedes the other by a *happened-before* relationship or they are causally concurrent. In the case of hash values, determining the causal relationships between two versions requires an examination of the (parent, child) pairs stored in the update log belonging to that object. In fact, to determine if a version causally precedes another version, given their hash values, the update log needs to be traversed in order to find a directed path or (parent, child) pairs linking the first version to the other.

5.3 Update stabilization strategies

One important characteristic of an optimistic replication solution is the strategy it uses to determine when a tentative update should become a stable update. Most systems also refer to such decision as update commitment. According to the model presented in Section 2, this step corresponds to the coordination phase.

This option is strongly tied to the consistency guarantees that a replicated system provides regarding the stable value of each replica.

5.3.1 Stability condition

One simple approach is to commit an update whenever a certain condition concerning the state of the replica is met. One example of such condition would be to commit a tentative update whenever all the updates it depended on were already stable. This is the approach followed by the Gossip framework (Section 7) and ensures a causal ordering of the stable updates.

The main disadvantage of this solution is that it does not take into account any concurrent updates that haven't yet been received by the replica manager at the time of the commit decision. A direct consequence of this fact is that, if distinct concurrent updates which verify the stability condition arrive at two replica managers in a different order, they will be committed in that order. As a result, the stable replica values of the system are not guaranteed to have the same contents after receiving the same collection of updates. That is, the stable updates are not guaranteed to be totally ordered.

5.3.2 Primary commit scheme

For most applications, however, the requirement of having the same contents on every stable replica value holds. In this context, a solution might be to assign an individual replica manager with the responsibilities of update commitment.

Such special replica manager, the primary replica manager, would receive tentative updates just as a regular replica manager. Additionally, it is capable of, according to some logic, decide upon the final stable ordering to apply to such updates. That decision is then propagated to the remaining replica managers. The flow of such information is done using the same protocol that is employed to propagate tentative updates. The difference, though, is that updates are now marked as stable.

A primary commit scheme guarantees that the stable updates at every replica are totally ordered, which is an important requirement in most domains. One example is the Bayou replicated system (Section 8).

The main drawback of this approach is that is relies on a single point of failure, which is the primary replica manager. In fact, this update commitment closely resembles the pessimistic approaches of primary copy or tokens, and thus shares their disadvantages. Before being committed, every update has to be consumed by the primary replica manager. In the case of failure or network partitioning, update committing can be severely disrupted.

5.3.3 Voting

In order to attain a higher availability of the update commitment process, some alternative solutions have resorted to quorum based approaches.

Here, the goal is to eliminate the single point of failure of the primary commit scheme. Instead, update commitment now requires that a plurality of replica managers agrees on the stable order of a given set of updates. In the case of network partitions, a partition holding a sufficient number of replica managers to constitute a quorum is thus able to commit updates. Some solutions do not even require the quorum to be simultaneously present in the same network partition. One example is Deno (Section 10), which employs an epidemic voting scheme, hence enhancing the availability of the update commitment process.

One other important characteristic of this approach is that the decision of the final stable order is no longer delivered to a single entity. Under circumstances where replica managers do not trust each other to carry out such decision, a quorum solution is more appropriate. This is the case of Oceanstore [KBC+00].

5.4 Log truncation

A replica manager keeps updates in a log for two main reasons. Firstly, the tentative updates cannot be applied onto the stable replica value. Since it is tentative, it must not be executed until it becomes stable.

On the other hand, stable updates that have already been applied to the stable replica value may not have been received at every other replica managers. Such stable updates should be stored in the log so that they can be propagated to other replica managers that haven't yet received them.

The update log is, however, the main source of memory overhead in an optimistic replicated system. This is especially relevant in mobile environments, where the memory resources of mobile

devices are typically scarce. Therefore, a fundamental requirement is that such overhead is kept low. To achieve this objective, replica managers must be able to discard unnecessary updates in their replica's logs so as to keep acceptable log sizes.

5.4.1 Conservative log truncation

One common approach in conventional optimistic replication systems is to discard updates from the log whenever a replica manager knows that such updates are no longer necessary to ensure replica consistency. Such decision is typically taken by each replica manager based on a worst case estimate of the updates received at the remaining replica managers.

The Bayou (Section 8) and the Gossip framework (Section 7) are systems which employ this conservative approach. The first one allows any replica manager to discard updates provided that they are stable. The second example maintains a version vector table to store a worst case estimate about the updates received by every replica manager in the system. When an update is known to have been received by every replica manager according to the version vector table, it is safe to discard it.

5.4.2 Relaxed log truncation

In some situations, though, using a conservative approach for log truncation does not suffice to effectively reduce the storage overhead caused by the log. In particular, network partitions or failure of replica managers can cause situations where the condition for discarding an update is not verified until the network partition or the failure are repaired. If such events are to occur frequently, a conservative approach may rapidly reach a state where a replica cannot accept more updates because no more memory space is available for the log.

One alternative would be to discard updates using a relaxed estimate. For instance, [KWK03] proposes and analyzes using a simple aging method to estimate when a given update is no longer needed by other replica managers. When an update is stored for more than a certain time in the log, it can be discarded from the log.

Whenever an update is discarded from the log, a replica manager must ensure that it has been applied to its stable replica value. Otherwise, the update information might be lost if that replica manager was the only one holding the update. The problem arises when the update to be discarded is a tentative update.

In this case, the tentative update must be applied to the stable replica value. Consequently, such value ceases to be stable to become a *tentative replica value*. Hence, the system looses its ability to provide a stable view of the replicated object. Such ability may be later regained if the replica manager receives information that the every tentative updates which were applied to the replica value have become stable.

5.4.3 No log

One extreme approach is not to maintain any update log. This might be seen as a relaxed log truncation approach in which an update log would be immediately applied to the replica value and then discarded.

Such solutions typically have no notion of update stability. Therefore every update is tentative and each replica manager simply maintains a tentative replica value. The memory requirements of an update log are thus avoided at the expense of weaker consistency guarantees. Rumor and Roam (Section 9) are examples of replicated systems which employ this approach.

6 Coda

Conventional distributed file systems, such as NFS [Now89] and AFS [MSC+86], can also be regarded as replicated systems. Here, the replicated objects are files and directories that are replicated from the server's disks to client caches for better performance. These systems are based on a client-server architecture and employ pessimistic consistency policies.

Such design options are appropriate for operation over a fixed network, where the server infrastructure should be always accessible to the wired clients. Network partitions and server failures are exceptional events that are expected to occur rarely. In this context, the availability issue can be traded by stronger consistency policies that ensure file system semantics closer to or equivalent to

those of local file systems. The popularity and wide use of these systems in fixed network domains is a symptom of their effectiveness.

However, when one allows the possibility of mobile clients, network partitions between such clients and the server infrastructure are no longer negligible [JHE99]. To achieve a better availability in the presence of such potentially disconnected clients, the Coda distributed file system [Sat02] enables clients to access the files stored in cache while being disconnected from the server machines [KS91].

In addition to disconnected operation at clients, Coda also enables optimistic read-write server replication in order to achieve a better fault-tolerance in the face of server failures. Since such mechanism can be regarded as an extension of disconnected operation, it will not be further discussed in the paper.

6.1 Disconnected Operation

Coda inherits most of AFS's design options, including whole-file caching. Under normal operation, clients are connected to the server infrastructure and AFS's pessimistic cache consistency protocol is employed. When a client becomes disconnected from the servers, however, it adopts a distinct mode of operation. An optimistic consistency strategy is then used to enable the disconnected client to read and update the contents of the locally cached files and directories. A user can thus work on the documents cached on his disconnected mobile device while he is away from his wired desktop PC.

A client, or *Venus* in AFS terminology, can be in one of three distinct states throughout its execution: *hoarding*, *emulation* and *reintegration*. The client is normally in the hoarding state, when it is connected to the server infrastructure and relies on its replication services. Upon disconnection, it enters the emulation phase, during which update operations to the cached objects are logged. When a connection is again available, the reintegration occurs, in which the update log is synchronized with the objects stored in the servers' disks. The hoarding state is then entered.

6.2 Hoarding

A crucial factor on the effective availability that is achieved by disconnected operation has to do with the set of objects that are cached at the moment when disconnection happens. If, at the moment of disconnection, the set of cached files does not include those files that the mobile user will work on during the emulation phase, cache misses will occur, therefore disrupting normal system operation. If, otherwise, the set of cached files contains most of the files that the user will access in the future, disconnected operation can successfully achieve the desired availability.

For the purpose of selecting the set of files that should be cached during the hoarding phase, in anticipation for a possible disconnection, Coda combines two distinct strategies. The first is based on implicit information gathered from the recent file usage, by employing a traditional *least recently used* cache substitution scheme. Complementarily, explicit information is used from a customized list of pathnames of files of interest to each mobile user, stored at a *hoard database*. Those files have an initial cache priority that is higher than the remaining cached files, in order to meet users expectations stated in the hoard database.

6.3 Replica State

Recalling the general replica state model described in Section 5.1, a simple mapping can be made to the state maintained at each Coda client in the emulation phase.

Namely, the stable value corresponds to the cached contents of files, obtained from the servers during the hoarding phase. The update log contains only the tentative updates that are issued at that client during disconnection. Log truncation is made at the end of a successful reintegration phase and completely deletes the contents of the log.

6.4 Conflict detection and resolution

One consequence of disconnected operation is that concurrent updates to different replicas that occur while some of those replicas are disconnected can lead to inconsistencies. This raises the problems of conflict detection and resolution that characterize any optimistic replication approaches. Such problems are handled during the reintegration phase.

Being a file system Coda has to deal with two basic types of replicated objects: directories and files. In contrast with files, directories have well

known semantics which can be exploited by the file system to automatically maintain their integrity in the face of conflicting updates at different replicas. Coda solves the problem of directory inconsistency by using a semantic approach that detects and automatically resolves conflicts. An example of such conflicts is when two disconnected clients each create new files on a common replicated directory. This will cause a conflict when the reintegration phase of both clients happens, since the directory replicas at each client have been concurrently updated. Using semantic knowledge, however, Coda easily solves this conflict by including both newly created files on the merged directory value.

In the case of files, though, no semantic knowledge is available about their contents. For this reason a syntactic approach, based on version vectors, is adopted. Each replicated file is assigned a *Coda Version Vector* (CVV), which is a version vector with one element for each server that stores that file.

When a modified file is closed by a client, each available server holding such file is sent an update message, containing the new contents of the file and the CVV currently held by that client, CVV_u. Each server i checks if $CVV_u \geq CVV_i$, where CVV_i is the CVV held by server i for the modified file. If so, the new update is applied at the server's replica and its associated CVV is modified to reflect the set of servers that successfully applied the update. If, otherwise, a conflict is detected at some server, that file is marked as *inoperable* and its owner is notified. If the client is operating in disconnected mode, this procedure is postponed until being performed during the reintegration phase.

6.5 Conclusions

Disconnected operation as defined by Coda still depends strongly on the server infrastructure, since the updates made during disconnection will only be available to other clients after reconciliation with the server. This may be acceptable in cases where file sharing between mobile clients are rare situations or disconnection periods are short and infrequent. However, expected ad-hoc networking scenarios suggest that co-present collaborative activities in the absence of a fixed infrastructure should occur frequently.

In those cases, Coda's disconnected operation model is inadequate, since the updates made in disconnected mode must be first propagated to the server. Thus, each mobile client within an ad-hoc network without access to the server infrastructure would still act like an isolated file system client. Probably, the updates made during the meeting would only be propagated after each participant had arrived at his office and synchronized his mobile device with his desktop PC connected to the wired infrastructure.

7 Gossip Framework

Ladin et al. [LLSG92] propose a framework for providing high availability replication services for applications with weak consistency requirements.

The Gossip framework considers three possible update ordering modes: causal, forced and immediate. Causal mode causes updates do be ordered according to the happened-before relationship (Section 3.2.2).

Forced and immediate updates are guaranteed to be total and causally ordered. The difference between both is that forced updates are causal and totally ordered with respect to other forced updates, but only causally ordered within the remaining updates. In contrast, immediate updates are causal and totally ordered against all updates of all modes.

The choice of which ordering mode to use is assigned to the applications that issue each update. However, the two stronger ordering modes have significant costs on the effective availability of the replicated system. They require the replica manager to belong to a network partition where a majority of the remaining replica managers is accessible, which may be a significant limitation in poorly connected environments.

A relaxed consistency protocol among the replica managers guarantees that all replica managers eventually receive all updates. Conversely, those updates are eventually applied at each replica, according to the ordering requirements associated with each update. Gossip's consistency protocol is based on version vectors (Section 5.2.1).

The primary purpose of the Gossip architecture is to provide highly available services through the use of causal updates. For its relevance, the remaining subsections will only concentrate on such operation mode.

7.1 Replica state

Each replica manager contains a replica state which is similar to that proposed in Section 5.1.

Two additional components are worth being mentioned: an *executed operation table*, which contains the identifiers of the updates that have already been executed by the replica manager; and a *timestamp table*, which maintains the stable value timestamps for each replica manager in the system.

The first structure is used to prevent updates from being applied more than once. The second structure is used for the purposes of log truncation and for determining the set of updates to be propagated to the other replica managers, as will be described ahead.

A conservative approach for log truncation is employed, based on the timestamps stored in the *timestamp table*. According to such information, the updates which are guaranteed to have been received by all replicas are discarded from the log.

7.2 Conflict detection and resolution

Consistency between replica managers is achieved by the propagation of *gossip messages* between replica manager pairs. Gossip messages contain information about the updates known to a sending replica in order to bring the receiving replica up to date.

A gossip message has two components: the tentative replica timestamp and a relevant portion of the log of the replica manager which is sending the gossip message. The information in the *timestamp table* regarding the target replica manager is used to provide an estimate of the set of updates that must be sent in the gossip message. It is a worst-case estimate, since the receiving replica manager may have received more updates by the time the gossip message is sent.

The Gossip consistency model ensures that the stable causal updates are applied to the replica stable value according the partial order defined by the happened-before relationship. In such ordering mode, dealing with concurrent updates is straightforward: each replica manager adds the concurrent updates to its replica's update log in the same order as it has received them. This means that different replicas may reach distinct values after receiving the same set of updates.

7.3 Additional consistency guarantees

Clients of the replicated system can access any available replica manager to issue read and write requests. Since relaxed consistency is employed between replica managers, a client can perform requests at replica managers with different replica values. Therefore it is possible that a client contacts a replica manager whose replicas reflect less recent values than other replica managers which have already been read by the client. To prevent such inconsistent situations, the Gossip framework ensures that each client obtains a consistent service over time.

Each client maintains an associated version vector, *prev*, that reflects the latest version which was accessed for a certain replica. Such version vector is supplied along with every request that is made at a replica manager.

In the case of a read operation, the contacted replica manager only returns the desired data value when the version vector which was sent along with the request, $read.prev$, causally precedes or is equal to the stable value timestamp of the replica manager, $stableTS$. That is, if $read.prev \leq stableTS$.

This condition guarantees that, if a client wishes to retrieve data from a replica that is older than other replicas from which it has already read from, the corresponding replica manager will hold back the result until the condition is fulfilled. This ensures that clients obtain a consistent service over time. Such condition is equivalent to the *monotonic reads session guarantee* which is referred in Section 8.3 in the context of the Bayou replicated system.

In the case of write operations, a similar scheme is used. A logged update, u can only be applied upon the stable replica value when Gossip's stability condition is verified: $u.prev \leq stableTS$, where $stableTS$ is the current replica stable timestamp. This condition states that, for an update to become stable, all the updates on which it depends on must have already been applied to the stable replica value. This is conceptually similar to the *monotonic writes session guarantee* (Section 8.3), defined in the context of the Bayou replicated system.

Along with the response to a read or write request, a new version vector is returned. In the case

of read requests, the new version vector reflects the accessed replica stable timestamp. If, otherwise, it was a write request, the returned version vector is the version timestamp which was assigned to that update upon its reception by the replica manager. In either case, such version vector is then merged with the previous client *prev* version vector.

7.4 Conclusions

The Gossip framework offers high availability replication services by enforcing causal ordering guarantees. Applications with relaxed correctness criteria are able to use the framework's services as long as any replica manager is available. Hence, the effects of network partitions or failure of some replica manager, which are likely to occur in mobile ad-hoc environments, are minimized.

The applications benefiting from such causal ordering guarantees must have relaxed correctness criteria. In particular, such applications have to be able to tolerate the absence of total ordering guarantees. In other words, the possibility of two replicas having distinct values after receiving the same set of updates must be regarded as correct. For semantical domains with more demanding correctness criteria, forced and immediate updates can be used at the cost of reduced availability.

The Gossip Framework requires each replica manager to maintain an update log. A conservative log truncation approach ensures that, when any two replica managers reconciliate, the logged updates at each of them will suffice to synchronize their replicas. Hence, transference of the complete replica value will never be required, which is an important feature when the size of the replicated objects is significative.

On the other hand, network partitions or failure of some replica manager may cause updates to remain indefinitely in a replica manager's log, since the unavailable replica managers have not yet received such updates. This means that, under such circumstances, the size of an update log can rapidly grow, as updates continue to be issued while older logged updates are not being discarded. For replica managers running at mobile devices, with typically poor memory resources, this is an important limitation. In fact, if a replica manager runs out of available memory for its update log, it can no longer receive update requests. Therefore, the system's availability is reduced.

8 Bayou

The Bayou System [DPS+94] is a mobile database replication system that provides high availability with weak consistency guarantees. Bayou employs a semantic approach which supports application-specific update conflict detection and resolution. For that purpose, Bayou's programming interface requires applications to provide conflict detection and resolution instructions along with each data update they make.

A Bayou client can issue updates at any accessible replica manager. Hence, a highly available service is provided. Replica managers exchange received updates in pairwise interactions called *anti-entropy sessions*.

Bayou's semantic approach to conflict detection and resolution markedly differentiates it from weak consistency systems whose consistency protocol is based on a syntactic scheme (such as Coda, Rumor, Roam or the Gossip framework). The expressiveness of the conflict detection and resolution instructions included in each Bayou update entrusts applications with stronger consistency guarantees than those provided by semantically-blind alternatives.

8.1 Replica state

Each Bayou replica manager contains a replica state which is similar to the one presented in Section 5.1.

8.2 Conflict detection and resolution

Bayou uses a dynamic variant of version vectors (Section 5.2.2) to syntactically identify and order replica versions. Tentative updates are epidemically propagated between replica managers and stored at each one's update log, ordered by the causal order defined by the version vectors of each update. Additionally, causally concurrent updates are totally ordered according to the identifiers of the replica managers that accepted such update from clients. A total and causal ordering of updates is therefore guaranteed.

A semantic strategy, however, complements the syntactic consistency scheme. Every update con-

tains conflict detection and resolution instructions, respectively designated as *dependency checks* and *merge procedures*, specified by the application which issued the update. Such components of an update should reflect the issuing application's semantics.

A replica manager executes an update's dependency check before applying it. A dependency check contains a query condition which is able to examine any part of the replicated database to determine whether a conflict exists or not. For instance, an update which books an appointment in a schedule database could check if, at the intended hour, any other event is already filling that time slot.

If the dependency check's query detects a conflict, the update's merge procedure is called. Such procedure alters the effect of the update. In the booking example, a merge procedure could rearrange the booking specification so that it would move to a nearby available time slot. Or, if no suitable alternative is found, the merge procedure can simply decide to abort the effect of the update.

Being an optimistic replicated system, it is possible that a replica manager receives, through anti-entropy with other peers, updates which are older than some of the tentative updates in the log. Therefore, such updates are placed in its correct position in the causal ordering, according to their version vector. In such cases, conflict detection and resolution need to be performed, not only upon the newly received updates, but also upon those updates that follow them in the log.

One important aspect is that merge procedures are deterministic. This means that, if two replica managers receive the same set of updates and order them equally, their tentative replica values will be the same.

8.3 Session guarantees

To accommodate for applications with stronger consistency requirements, Bayou allows additional consistency guarantees to be selected by each application. Such consistency guarantees are designated as *session guarantees* [DBTW94].

A *session* is an abstraction for a sequence of read and update operations performed on a database during the execution of an application. When a session guarantee is selected by some application, Bayou ensures that the sequence of operations within the session's duration will meet the consistency requirements imposed by the session guarantee. Four session guarantees are offered: *Read Your Writes*, *Monotonic Reads*, *Writes Follow Reads* and *Monotonic Writes*. A detailed specification of each session guarantee can be found in [DBTW94]. That is the trade-off for the increased consistency.

In order to enforce each of the four session guarantees, Bayou restricts the set of replica managers that can be contacted by each client to issue an operation request. Given a particular operation request, it can only be delivered at those replica managers where reception of the request won't violate the currently selected session guarantees. For this reason, requesting a session guarantee can have a harmful impact on the availability of the replicated system.

8.4 Update stability

Bayou adopts a hybrid replication strategy, in which an eventually sequential consistent stable value of the replicated data is also available, in addition to the tentative value. A *primary commit scheme* (Section 5.3.2) is used in order to determine the stability of its updates [TTP+95].

In this scheme, a replica manager is designated as the primary replica manager. The primary replica manager takes responsibility for defining the final ordering of the tentative updates as it receives them. Those updates then become stable updates. Such ordering information is then propagated to the other replica managers by anti-entropy.

When a replica manager receives information about an update that has become stable, that update is inserted at the head of the stable portion of the log. If its corresponding tentative update is present in the update log, it is removed from there.

In response to the reordering of the update log caused by the arrival of the new stable update, all the tentative updates that succeed it in the log must be undone. The new stable update is then applied, which includes performing its conflict detection and resolution procedures. Finally, all tentative updates that follow it are re-applied.

Update log truncation is dependent in the commit scheme, since a replica manager is allowed to discard updates from its log as long as those updates have become stable [TTP+95].

8.5 Conclusions

Bayou's semantical approach to conflict detection and resolution makes replication non-transparent to the applications. As a result, Bayou can exploit such semantical knowledge to provide consistency guarantees that are stronger than the causal consistency guarantees of the Gossip Framework (Section 7).

As a drawback, application programmers are now responsible for supplying dependency checks and merge procedures. Depending on the application's semantics, the amount of conflicts that need to be detected, along with their corresponding possible resolution actions, may severely increase the programming complexity. Furthermore, some application semantics may include complex conflicts for which an appropriate resolution procedure may require an external decision from the user. Since merge procedures must be deterministic, such user input is not allowed. The effectiveness of Bayou's semantical approach is, therefore, restricted to application domains with underlying data semantics of relative simplicity.

On the other hand, Bayou's users must be willing to deal with having the results of their data operations changing over time. This can occur if users read the tentative value of a replica after issuing some update, which may later have its effect altered by its merge procedure as a result of the receipt of some conflicting update [TDP+94].

Finally, Bayou's performance is significantly constrained by the overhead resulting from the application of dependency checks and merge procedures. In particular, when an update is received by a replica manager and inserted in the middle of the update log, the dependency checks, and possibly merge procedures, of the updates that follow it in the log must be re-applied.

9 Roam

Roam [DRP99] is a optimistically replicated file system intended for use in mobile environments.

Roam allows any replica manager to serve operation requests, without the need of accessing a centralized server. This contrasts with Coda's model of disconnected operation (Section 6), in which all updates must be propagated first to a server machine that further propagates them to other clients.

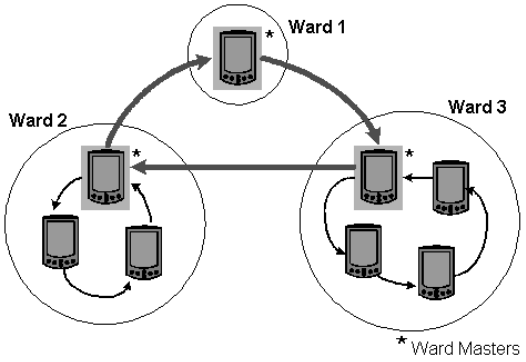

Figure 8: An example of a simple ward configuration. Reconciliation amongst Roam's replica managers is based on this topology.

Therefore, system operation is not dependent on the availability of a server infrastructure. Instead, any pair of mobile peers can exchange replica updates while being disconnected to the fixed network.

Roam is an extension to the Rumor file system [GRR+98], which in turn borrowed much of the replica consistency mechanisms from the Ficus file system [GHM+90]. Roam operates at application level, relying on the local file system services to store each replica.

Gossip messages propagate information about the updates that each replica has received so far to the remaining replicas. Replica consistency is achieved by performing periodic reconciliation sessions between peers in which gossip messages are exchanged.

9.1 Ward model

Roam's architecture focuses on providing improved scalability by adopting a hierarchical reconciliation topology, designated as the *ward model* [RRPK01]. It groups nearby peers into domains called *wards* (wide area replication domains), according to some measure of proximity. Each ward has an assigned ward master. The result is a two-level hierarchical topology (Figure 8).

Reconciliation amongst wards is performed ex-

clusively between ward masters. Inside each ward, all its ward members are configured into a conceptual ring. Such ring topology imposes that each ward member reconciles only with the next ring member. The ring is adaptive, in the sense that it reconfigures itself in response to changes in the ward composition. Therefore, enhanced scalability is achieved.

9.2 Replica state

The replica state in Roam differs significantly from the general model that was introduced in Section 5.1. For each replicated file, a replica manager only stores the replica value and its associated version identifier. No update log is maintained.

Roam's consistency model does not consider the notion of update stability. Thus, the replica value that is stored at replica managers reflects its tentative value.

9.3 Conflict detection and resolution

As with Coda, directory conflicts are automatically resolved by taking into account their well know semantics. In the case of files, the approach of dynamic version vectors is employed. Each file replica is assigned a dynamic version vector and the consistency protocol ensures strict causal guarantees.

Periodically, each replica manager reconciliates its replicas with another replica manager, according to the reconciliation topology imposed by the ward model. The reconciliation procedure itself is divided into three phases: *scan*, *remote-contacting* and *recon*.

During the scan phase, the replica manager examines its file replicas to check for new versions and, when necessary, update their version vector. This is done by a simple comparison of the modification times of each file replica, which are obtained by querying the local file system. If the current modification time of a file replica is greater than that obtained during the last reconciliation, then a new version exists and the replica's version vector is updates to reflect it.

In the second phase, a remote replica manager is contacted, according to the current ward topology, and asked for perform a similar version detection on its set of file replicas. As a result, the list of the file replicas and corresponding version vectors of the remote replica manager is sent back to the requesting replica manager.

Finally, version vectors of the files that are mutually replicated by both replica managers are compared during the recon phase. If the remote version of a file dominates the local version, its entire contents are transferred from the remote peer to update the local replica value. If, otherwise, the local version dominates the remote version, no action is taken since reconciliation in Roam is a one-way operation.

There is support for integration of type-specific file reconciliation tools into the conflict resolution mechanisms. In the case of version conflicts regarding types of files for which a reconciliation tool has been installed, a replica manager automatically resolves the conflict and avoids notifying the user. Otherwise, the user is notified by email of the conflict and no automatic resolution is performed.

9.4 Conclusions

The Roam replicated file system provides a serverless service, intended for mobile networks. By using an optimistic approach, any mobile replica manager is able to accept operation requests, which allows for enhanced availability.

Replica consistency relies on an epidemic propagation of updates between replica managers. Roam adopts a two level hierarchical topology in which replica managers are grouped into wards. Update propagation is achieved by reconciliation sessions between pairs of replica managers within each ward and between ward masters. The result is increased scalability. Such architectural model, which dynamically adapts itself to reflect the proximity amongst replica managers, is particularly effective in supporting scenarios where ad-hoc co-present groups of mobile devices are frequently formed.

To allow for a dynamic set of replica managers, Roam uses dynamic version vectors to timestamp replica versions. Section 5.2.3 discusses this approach in greater detail.

The consistency protocol itself is characterized by its relative simplicity. The absence of an update log avoids the need to keep track of each update issued to the replicated file system objects. Instead, new replica versions are detected only at periodic reconciliation phases by analysis of their modification times. Moreover, since no update log

is maintained, Roam imposes a low memory overhead. Such factor is especially adequate for mobile devices, which typically have poor memory resources.

A significant drawback is the consistency guarantees that Roam provides. Since no notion of stability exists, every read request that a client may issue will only return tentative data. This aspect restricts Roam's applicability to applications whose correctness criteria are sufficiently relaxed to tolerate dealing with tentative data.

10 Deno

Deno [Kel99] is a replicated database protocol designed for use in mobile and weakly-connected environments. It adopts an optimistic strategy by allowing updates to be received at any replica manager in order to provide a highly available service.

Replica consistency is achieved through an epidemic update propagation scheme. Being an optimistic strategy, conflicting tentative updates can be accepted at distinct replicas. Although the requirements of certain applications allow them to access the possibly inconsistent, tentative values of replicas, other application domains require stronger consistency guarantees. For such applications, the only safe value to access is the stable value of a replicated object.

Deno's consistency protocol is responsible for reaching a consensus amongst all replica managers on which one of such conflicting, tentative updates is to be committed. A distinguishing feature of Deno's design is that it relies on an voting approach to achieve such goal.

This strategy eliminates the problem of a single point of failure that characterizes primary commit schemes (Section 5.3.2). To deal with frequent disconnection, voting is performed by means of a pairwise epidemic protocol. The following sections further analyze Deno's consistency protocol.

10.1 Elections

Deno regards update commitment as a series of elections. Each election decides, amongst a collection of concurrent tentative updates, which one of them should be accepted as stable while the remaining updates are aborted. Each replica manager acts as a voter in such elections. Similarly, each tentative update acts as a candidate for one election. Once an election is over, a new election is started.

Each replica manager, or voter, has an assigned *currency*, which determines that replica manager's weight during each voting round. An invariant of the system is that there is a fixed amount of currency, 1.0, which is divided among all replica managers of a given object. Currencies are not necessarily distributed uniformly among replica managers. Neither is their distribution static, since replica managers can perform currency exchange operations.

Voting takes place in a decentralized manner, relying on epidemic information propagation between voters to reach an eventual election result. The result of each election is, thus, determined individually by each voter in function of the information propagated from its peers. A voter decides that an election has terminated and update u_j has won it when:

(1) total currency voted so far in favor of $u_j > 0.5$

or

(2) for each other candidate update, u_k, (total currency voted so far in u_k + total currency of votes not yet cast) < total currency voted so far in u_j

Election information flows from voter to voter by unidirectional interactions between voters, designated as anti-entropy sessions. In each anti-entropy session, one voter propagates the voting information he is aware of to another voter. Such information includes (1) the elections that have already terminated and their outcomes, and (2) the votes that have been cast in the current election. The receiving voter then updates his election state according to such information. Additionally, if the receiving voter hasn't yet cast a vote, he automatically votes on the same candidate which has been voted by the voter which initiated the anti-entropy session.

Whenever any of the election termination conditions described above is verified by a replica manager, it decides by self initiative that such election is over and the winning candidate is acknowledged

as a committed update. This style of information flow causes situations in which the replica managers have different views of the current election state. However, the epidemic propagation by anti-entropy sessions ensures that all replica managers will eventually reach the same state for every election that occurs.

10.2 Replica state

A replica manager stores, for each replica, a state that is similar to that proposed in Section 5.1. It should be noted that the tentative updates in the log constitute, in fact, a queue of candidate updates waiting to be voted.

At a given moment, if a replica manager has a non-empty set of tentative updates, then the first of such updates is currently being voted in the election that is taking place. It is competing against concurrent tentative updates of other replica managers.

10.3 Conflict detection and resolution

The voting process implicitly deals with the tasks of conflict detection and resolution. When a tentative update is issued at a given replica manager without any pending tentative updates, it is automatically placed as a candidate of the current election. Additionally, the replica manager votes on that candidate update.

If a conflict occurs it means that another concurrent tentative update was issued at a different replica manager. In such case, both conflicting updates will be rival candidates in the election.

Deno's conflict resolution follows the strategy of selecting only one of the conflicting updates and aborting the remaining ones. The epidemic voting scheme Deno ensures that only one candidate wins the election and such winner is the same at every replica manager. Hence, conflict resolution is accomplished when a election terminates.

10.4 Conclusions

Deno provides highly available replication services, designed for use in mobile and weakly connected environments.

It follows an optimistic replication strategy that offers strict causal ordering of the tentative updates received at each replica. Applications with relaxed consistency requirements can access such tentative replica state. In complement, an eventually sequentially consistent stable value is also offered to applications with more demanding correctness criteria.

The main contribution of Deno's protocol design is the adoption of an epidemic voting scheme for update commitment. In contrast to a primary commit scheme, a voting scheme is no longer reliant on a single replica manager. Instead, a quorum of replica managers is required to decide which update, amongst a collection of conflicting updates, is to be committed. As a result, higher availability is achieved for the update commitment scheme.

The performance and network usage overheads inflicted by the voting scheme are its main disadvantages. In fact, under situations of high connectivity amongst replica managers, defining one of such replica managers as a primary server yields significantly faster update commitment and less network communication than using Deno's voting scheme [Kel99]. This is an expected consequence of the centralized nature of the primary server scheme, in contrast with the voting approach.

11 AdHocFS

AdhocFS is a distributed file system designed for supporting pervasive computing in mobile ad-hoc environments. The system is based on the premiss that, on such scenarios, a fixed server infrastructure that provides the file system's services may not always be available. Therefore, AdHocFS's goal is to effectively support information replication between mobile users in the absence of such infrastructure.

Furthermore, AdHocFS distinguishes between situations where mobile devices are working in isolation from any other devices and situations where groups of mutually accessible mobile devices cooperatively share and manipulate information. To deal with the distinct characteristics of each of such scenarios, AdHocFS uses distinct replica consistency strategies for each case, as will be described in the following subsections.

11.1 Home Servers and Ad-Hoc Groups

AdHocFS's architecture considers the existence of a trusted stationary server infrastructure and a collection of wireless mobile devices that may frequently be disconnected from such infrastructure.

Each file has a replica stored on a replica manager located on the server infrastructure, which is designated as the *home server* for that file.

Mobile devices can obtain a replica of the files stored at a home server when a connection is available. Upon disconnection, mobile devices tentatively operate upon the local replicas using a optimistic replication strategy.

When disconnected from the fixed infrastructure, mobile devices constitute *ad-hoc groups*. An ad-hoc group is a collection of mobile devices which are mutually accessible within one-hop wireless links. One extreme case is a singleton group, in which a group is formed by only one isolated device. AdHocFS dynamically manages the membership changes of ad-hoc groups as they are merged or separated.

This ad-hoc group model shares some similarities with the ward model of the Roam replicated system (Section 9). However, one key aspect differentiates it from the ward model. In the case of the latter, replica managers are grouped with the sole intention of achieving better scalability. On the other hand, AdHocFS explores the high connectivity that exists among the replica managers within an ad-hoc group for consistency purposes.

A pessimistic replica consistency approach is used within the members of each ad-hoc group, complementing the general optimistic consistency protocol. As a result, sequential consistency is accomplished if one only regards the set of replicas located in each ad-hoc group. The next subsections describe these issues in closer detail.

11.2 Replica state

Each home server stores file replicas, along with a scalar timestamp for each of them. Such timestamps identify the current file versions stored at the home server. Each time an update is applied at the replica at the home server, upon request by a mobile replica manager, the timestamp is incremented. Regarding the model in Section 5.1, the replica value at the home server represents the stable value of the replicated file; conversely, the timestamp associated with that replica stands for the stable value timestamp.

When connected to the trusted infrastructure, mobile devices can obtain a stable replica of a given file, along with its associated stable value timestamp, by contacting its home server.

The replica value copied from the home server to a replica manager initially reflects the stable value of that replica, as stored in the home server. However, such value is actually used as the tentative value of the replica, since the tentative updates that are successfully received at a replica manager are immediately applied to it.

Associated with each replica, a replica manager maintains a log where it stores information about the tentative updates that are issued during periods of disconnection from the home server. The log maintained by AdHocFS's replica managers is different from the update log description introduced in Section 5.1. For each update, the corresponding entry in the log is complemented with information about the members that constituted the ad-hoc group where the update was issued. On the other hand, information about the updates itself simply contains the file blocks' addresses (instead of their actual contents) which were affected by the update. No additional update specification information is stored in the log.

One important aspect is that no timestamps are assigned to the updates referenced in the log. The only information regarding the ordering of the updates is their actual ordering in the log. This representation of the update sequence is normally designated as a *causal history* [SM94].

Using AdHocFS terminology, the stable value timestamp and the log constitute a *Coherency Control List*, or *CCL*.

11.3 Conflict detection and resolution

To allow mobile devices to operate while disconnected from the home servers of each file, AdHocFS uses an overall optimistic replication strategy. This way, read and update requests can be served, albeit tentatively, by each mobile replica manager when the home servers are not available.

In such situations, any update that is issued to a given replica is stored in its update log, along with membership information about the current ad-hoc group of the replica manager. For instance, if update $u1$ is requested at replica manager A when it is in a group formed by replica managers A, B and C, the tuple $< u1, \{A, B, C\} >$ is added to the log.

The generic consistency protocol relies on the *prefix relationship* between CCLs. A CCL h_1 is a prefix of CCL h_2 if and only if: (1) their stable value timestamps are equal and (2) the list of tuples $< update, ad - hocgroupcomposition >$ of h_1 is a prefix of the same element of h_2. Whenever the CCL of a certain replica is a prefix of another replica's CCL, the latter causally dominates the former. In that case, synchronization can be performed in a straightforward manner by propagating the remaining tuples from the dominant replica's CCL to the other one.

When a mobile replica manager contacts the home server of a given file to synchronize its replica value, the generic consistency protocol is performed between them, with one difference: the timestamp comparison to determine the prefix relationship is not considered. If the home server's stable replica dominates the mobile replica manager's, it sends the missing updates to the mobile replica. If, instead, the replica at the mobile replica manager is dominant, the home server receives the new updates and accordingly applies them to its replica stable value and increments its timestamp. In either cases, the replica manager finally substitutes the stable timestamp in the replica's CCL by the current timestamp, received from the home server.

On the other hand, mobile replica managers can also synchronize their mobile replicas. All such pairwise synchronization sessions occur within the ad-hoc group, between its members.

Within an ad-hoc group situation, a stricter consistency protocol is used. In such situations, a pessimistic, single-writer multiple-readers token approach (Section 4.2) is employed so as to explore the high connectivity that is expected to exist amongst the ad-hoc group members.

When a replica manager receives a client request to access a logical file, it checks if it is holding the necessary token for the operation. If not, the token needs to be obtained from the current token holders. Obtaining a token requires the interested replica manager to synchronize its file replica with the replica stored by the replica manager that last held a write token to that logical file. Synchronization makes use of the generic optimistic protocol, based on the CCLs associated with the file replicas. When synchronization successfully finishes, the token is granted to the requesting replica manager, which is then able to perform the desired access to the replica.

It should be stressed out that the token approach is a pessimistic strategy only in the scope of an ad-hoc group. It ensures strong consistency amongst the replicas of each ad-hoc group, but not amongst every replica manager in the system. Consequently, on a global scope, an optimistic approach still prevails.

11.4 Conclusions

AdHocFS architecture is based on the existence of fixed server infrastructures where the home servers are located and where the stable replicas of files are accessible. The mobile devices act as replica managers and are able to optimistically work on the local replicas while in disconnection from the home servers.

Under such circumstances, an expected scenario is that of mobile devices working cooperatively on some shared files within an ad-hoc network. For the duration of the ad-hoc network, its replica managers benefit from high connectivity links to the other ad-hoc members. This fact is explored by AdHocFS by using a pessimistic strategy on the scope of one-hop ad-hoc groups.

In such scenario, updates from multiple users to the same shared files are expected to happen frequently as a result of the cooperative interactions. Using a flat optimistic approach with lazy update propagation, most of such updates would probably result in conflicting updates from the consistency protocol's point of view. This would happen if, before an update was propagated in a lazy fashion to every replica manager, a new update had already been issued at some replica manager that still hadn't received the previous update.

Using the token strategy of AdHocFS, issued updates are held back until a write token is granted to the replica manager. That, in turn, won't happen until all the preceding updates have been received by that replica manager. Therefore, consecutive updates issued in an ad-hoc group that would oth-

erwise be considered conflicting are now causally related by the consistency protocol.

It should be noted, however, that the global replication protocol is still optimistic if one considers every replica manager in the system and, consequently, weak consistency guarantees are offered.

Three main drawbacks can be identified in AdHocFS. Firstly, only the home servers provide access to the stable values of each file. The mobile replica managers hold only tentative replica values, where tentative updates are immediately applied upon being received. Hence, mobile users that are disconnected from the home server are only able to access the tentative version of files.

Additionally, the home servers act as the primary servers described in Section 5.3.2. As discussed in the same section, update commitment can be disrupted by network partitions or failure of the home servers.

Finally, a causal history approach is used for representing the tentative updates at each replica. This approach requires that, in order to determine the prefix relationship between two replicas, the contents of both replicas' CCLs must be examined. This typically compels one CCL to be entirely transferred to the other replica manager in order to determine if the prefix relationship is met. Such requirement is an important overhead in mobile environments, where network bandwidth is expected to be scarce.

12 Overall comparison and discussion

Tables 1 and 2 present an overall comparison between key design aspects of the surveyed systems. The main implications of such design decisions are summarized as follows.

Coda
Main Advantages:

- Effective, highly available solution for mobile or weakly connected systems based on a server infrastructure.
- Combined hoarding strategy uses implicit and explicit file usage information.

Main Disadvantages:

- Collaborative activities in ad-hoc networks disallowed: update propagation is dependent on server infrastructure.
- Storage requirements of update log when in disconnected mode.
- Applications must be willing to deal with having their updates aborted as a result of conflict resolution during reintegration phase.

Gossip Framework
Main Advantages:

- Optimistic approach for *causal updates* yields high availability with no need for a fixed infrastructure: only one replica manager has to be accessible and updates are exchanged between replica managers through an epidemic propagation protocol.
- Adaptability to a wide range of correctness criteria: causal, forced or immediate updates provide distinct consistency guarantees.

Main Disadvantages:

- High availability only provided for causal updates; forced and immediate updates require replica manager to belong to a majority partition.
- Applications' semantics must accept that an identical sequence of concurrent causal updates can produce distinct results in different replica managers.
- Conservative log truncation may require large memory resources if unavailability of some replica managers is frequent.

Bayou
Main Advantages:

- Optimistic approach yields high availability with no nedd for fixed infrastructure: only one replica manager has to be accessible and updates are exchanged between replica managers through an epidemic propagation protocol.
- Exploits semantical knowledge to provide stronger consistency guarantees.

	Coda	Gossip framework	Bayou
Type of replicated system	File system	Database	Database
Version identifiers	Version vectors	Version vectors	Bayou's version vectors
Replication strategy	Optimistic when in disconnected operation. Pessimistic, (primary copy) when connected, if optimistic server replication is not employed.	Optimistic	Optimistic
Update propagation architecture	Client-server	Peer-to-peer	Peer-to-peer
Use of object's semantics	Automatic conflict resolution for directories. Syntactic approach for files, by default. Installation of type-specific file mergers for conflict resolution of files is supported.	Pure syntactic approach for conflict detection and resolution.	Each individual update request comprises a dependency check and a merge procedure, supplied by the application.
Consistency guarantees	Strict causal ordering when in disconnected operation; sequential consistency when connected.	Causal ordering of updates, complemented with *monotonic reads* and *monotonic writes* guarantees.	Total and causal ordering of updates in tentative view, extended with semantic conflict detection and resolution. Eventual sequential consistency in stable view.
Commit scheme	Primary commit scheme: fixed servers determine stable updates.	Tentative update is considered stable when all updates that causally precede it have been received by the replica manager and applied to the stable value.	Primary commit scheme: designated replica manager determines stable updates.
Storage requirements	Update log and stable value (cached files).	Update log and stable value.	Update log and stable value.

Table 1: Overall comparison of surveyed systems (Coda, Gossip Framework and Bayou).

- Adaptability to applications with different correctness criteria, by allowing selection of tentative or committed view of replicated data.

Main Disadvantages:

- Non-transparent approach requires additional application programming complexity.
- Effectiveness of semantic approach restricted to simple application semantics.
- Users must be willing to deal with having the results of data operations changing over time.
- Performance overhead caused by application of dependency checks and merge procedures.
- Increasing replica manager identifier sizes due to recursive naming scheme.

Roam
Main Advantages:

- Optimistic approach yields high availability with no need for fixed infrastructure: only one replica manager has to be accessible and updates are exchanged between replica managers through an epidemic propagation protocol.
- Simplicity of consistency protocol, with low memory requirements due to absence of update log.
- Ward model provides increased scalability in situations where mutually accessible groups of replica managers are frequent.

Main Disadvantages:

- No notion of stability is offered.
- Applications must have sufficiently relaxed correctness criteria to tolerate dealing only with tentative data.

Deno
Main Advantages:

- Optimistic approach yields high availability with no need for fixed infrastructure: only one replica manager has to be accessible and

	Roam	Deno	AdHocFS
Type of replicated system	File system	Generic objects	File system
Version identifiers	Dynamic version vectors	Election counter	Home server timestamp + update log contents
Replication strategy	Optimistic	Optimistic	Optimistic
Update propagation architecture	Ward model	Peer-to-peer	Peer-to-peer epidemic update propagation within ad-hoc group members. Client-server relationship between mobile devices and home server.
Use of object's semantics	Automatic conflict resolution for directories. Syntactic approach for files, by default. Installation of type-specific file mergers for conflict resolution of files is supported	Pure syntactic approach.	Automatic conflict resolution for directories. Pure syntactic approach for files.
Consistency guarantees	Strict causal ordering of updates.	Strict causal ordering of updates in tentative view. Eventual sequential consistency in stable view.	Strict causal consistency with sequential consistency in the scope of an ad-hoc group. Eventual sequential consistency at replicas at home servers.
Commit scheme	No notion of stability.	Voting scheme with variable currencies.	Primary commit scheme: home servers determine stable updates.
Storage requirements	Tentative value.	Update log and stable value.	Tentative value and Coherency Control List.

Table 2: Overall comparison of surveyed systems (Roam, Deno and AdHocFS).

updates are exchanged between replica managers through an epidemic propagation protocol.

- Adaptability to applications with different correctness criteria, by allowing selection of tentative or committed view of replicated data.
- High availability of update stability scheme, by employing an epidemic voting protocol.

Main Disadvantages:

- Performance and network usage overhead in situations of high connectivity.
- Applications must be willing to deal with having their updates aborted as result of an election.

AdHocFS
Main Advantages:

- Optimistic approach yields high availability: only one replica manager has to be accessible and updates are exchanged between replica managers through an epidemic propagation protocol.
- Pessimistic consistency protocol employed in scenarios of co-present collaborative activities provides stronger consistency guarantees amongst the members of each ad-hoc group.

Main Disadvantages:

- Stable data is only available at home servers infrastructure; disconnected applications must tolerate manipulating tentative data.
- Update stability is disrupted by the unavailability of the home server.
- Causal history approach imposes network usage overhead for conflict detection.

13 Conclusions

The emergence of more powerful and resourceful mobile devices, as well as new wireless communication technologies, is turning the concept of ad-hoc networking into a viable and promising possibility for ubiquitous information sharing. For that purpose, this paper focuses on the issue of replication strategies for mobile ad-hoc environments.

The inherent characteristics of ad-hoc networks bring up new challenges for which most conventional replication systems don't provide an appropriate response. This paper exposes the main challenges of a replicated system operating in the particular environment of mobile ad-hoc networks. Based on a common conceptual model, some key design strategies were described and compared in terms of their adequacy to the mobile ad-hoc scenario. Finally, a survey of some relevant state of the art replicated solutions presented actual implementations of the main design alternatives.

As a conclusion from such analysis, some general design guidelines towards an effective replication solution for mobile ad-hoc environments can be drawn:

- High availability.

  The high topological dynamism of mobile ad-hoc networks entails frequent network partitions. Moreover, the possible absence of a fixed infrastructure means that most situations will require the services within the network to be offered by mobile devices themselves. Such devices are typically severely energy constrained. As a result, the services they offer are susceptible of frequent suspension periods in order to save battery life of the server's device. From the client's viewpoint, such occurrences are similar to server failures.

  These aspects emphasize the need for high availability replication services, so as to minimize the effects of expectable network partitions and device suspension periods. Pessimistic replication strategies are normally overly restrictive to fulfill such a requirement.

- Adaptability to different correctness criteria.

  Optimistic replication strategies offer high availability as a trade-off for consistency. While certain applications are able to benefit from such increased availability, some application semantics demand stronger consistency guarantees.

  In order to be adaptable to a wider set of applications, replicated systems should offer multiple consistency levels: from a relaxed consistency, highly-available to a sequentially consistent mode of replica access. Hybrid solutions, which allow applications to choose between the tentative and a stable views of data, are an example.

- Memory and bandwidth usage.

  Whichever strategy is taken, the memory and bandwidth limitations of mobile devices and wireless links, respectively, must be taken into account.

  For optimistic strategies, the update log is the main memory overhead and appropriate truncation schemes must be employed. Conservative log truncation schemes may become ineffective in the presence of network partitions or failures of some replica managers. Relaxed log truncation schemes should be considered when regarding the poor memory resources of mobile devices.

  Moreover, network usage is typically dominated by replica synchronization. Maintaining an update log allows for an incremental replica synchronization, by transferring only the necessary updates instead of an entire replica.

- Support for operation under ad-hoc groups.

  An important usage scenario of ad-hoc networking is that of mutually accessible mobile devices working cooperatively within an ad-hoc network. In such ad-hoc groups, replica managers benefit from high connectivity links to the other group peers.

  Accordingly, users expectations may assume strong consistency guarantees due to the accessibility amongst the ad-hoc group members. Therefore, pessimistic replication strategies should be employed on the scope of each ad-hoc group.

References

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the International Conference on Software Engineering*, 1976.

[BI03] M. Boulkenafed and V. Issarny. Adhocfs: Sharing files in wlans. In

Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications, Cambridge, MA, USA, April 2003.

[DBTW94] K. Petersen M. J. Spreitzer M. M. Theimer D. B. Terry, A. J. Demers and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, September 1994.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3):341–370, 1985.

[DPS+94] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[DRP99] P. Reiher D. Ratner and G. Popek. Roam: A scalable replication system for mobile computing. In *Mobility in Databases and Distributed Systems*, 1999.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[GCK01] Jean Dollimore George Coulouris and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education 2001, 3 edition, 2001.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *Proc. 1990 Summer USENIX Conf.*, Anaheim, June 11-15 1990.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles SOSP 7*, pages 150–162, Asilomar Conference Grounds, Pacific Grove CA, 1979. ACM, New York.

[GRR+98] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[JHE99] Jin Jing, Abdelsalam Sumi Helal, and Ahmed Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2):117–157, 1999.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[Kel99] P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, 1999.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. Association for Computing Machinery SIGOPS, October 1991.

[KWK03] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 229–239. ACM Press, 1986.

[LH98] Paul Luff and Christian Heath. Mobility in collaboration. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, pages 305–314. ACM Press, 1998.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[LS90] Eliezer Levy and Abrahram Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990. TR-89-04 mar. '89 54 pages.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[MSC+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[MW82] T. Minoura and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database systems. *IEEE Transactions on Software Engineering*, SE-8:173–189, 1982.

[Nat95] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

[Now89] Bill Nowicki. Nfs: Network file system protocol specification. Internet Request for Comment RFC 1094, Internet Engineering Task Force, March 1989.

[PSTT96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[Rat97] David Howard Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing. Technical Report 970044, 31, 1997.

[RD01] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[RPC+93] P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek. Truffles—a secure service for widespread file sharing, 1993.

[RRPG99] David H. Ratner, Peter L. Reiher, Gerald J. Popek, and Richard G. Guy. Peer replication with selective control. *Lecture Notes in Computer Science*, 1748, 1999.

[RRPK01] David Ratner, Peter L. Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[Sat02] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[Shr02] Brooke Shrader. A proposed definition of 'ad hoc network'. May 2002.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, 1979.

[Tan95] A.S. Tanenbaum. *Distributed operating systems*. Prentice Hall, London, 1995.

[TDP+94] Marvin Theimer, Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, and Brent Welch. Dealing with tentative data values in disconnected work groups. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265:94–104, 1991.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Technical Commitee on Distributed Processing.

[YV] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. pages 305–318.