Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.

Abstract

Bayou is a replicated, weakly consistent storage system designed for a mobile computing environment that includes portable machines with less than ideal network connectivity. To maximize availability, users can read and write any accessible replica. Bayou's design has focused on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system, ensuring that replicas move towards eventual consistency, and defining a protocol by which the resolution of update conflicts stabilizes. It includes novel methods for conflict detection, called dependency checks, and per-write conflict resolution based on client-provided merge procedures. To guarantee eventual consistency, Bayou servers must be able to rollback the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

1. Introduction

The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. The advent of mobile computers, in the form of laptops and personal digital assistants (PDAs) enables the use of computational facilities away from the usual work setting of users. However, mobile computers do not enjoy the connectivity afforded by local area networks or the telephone system. Even wireless media, such as cellular telephony, will not permit continuous connectivity until per-minute costs decline enough to justify lengthy connections. Thus, the Bayou design requires only occasional, pair-wise communication between computers. This model takes into consideration characteristics of mobile computing such as expensive connection time, frequent or occasional disconnections, and that collaborating computers may never be all connected simultaneously [1, 13, 16].

The Bayou architecture does not include the notion of a "disconnected" mode of operation because, in fact, various degrees of "connectedness" are possible. Groups of computers may be partitioned away from the rest of the system yet remain connected to each other. Supporting disconnected workgroups is a central goal of the Bayou system. By relying only on pair-wise communication in the normal mode of operation, the Bayou design copes with arbitrary network connectivity.

A weak connectivity networking model can be accommodated only with weakly consistent, replicated data. Replication is required since a single storage site may not be reachable from mobile clients or within disconnected workgroups. Weak consistency is desired since any replication scheme providing one copy serializability [6], such as requiring clients to access a quorum of replicas or to acquire exclusive locks on data that they wish to update, yields unacceptably low write availability in partitioned networks [5]. For these reasons, Bayou adopts a model in which clients can read and write to any replica without the need for explicit coordination with other replicas. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions.

Unlike many previous systems [12, 27], our goal in designing the Bayou system was *not* to provide transparent replicated data support for existing file system and database applications. We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application.

To this end, Bayou provides system support for application-specific conflict detection and resolution. Previous systems, such as Locus [30] and Coda [17], have proven the value of semantic conflict detection and resolution for file directories, and several systems are exploring conflict resolution for file and database contents [8, 18, 26]. Bayou's mechanisms extend this work by letting applications exploit domain-specific knowledge to achieve automatic conflict resolution at the granularity of individual update operations without compromising security or eventual consistency.

Automatic conflict resolution is highly desirable because it enables a Bayou replica to remain available. In a replicated system with the weak connectivity model adopted by Bayou, conflicts may be detected arbitrarily far from the users who introduced the conflicts. Moreover, conflicts may be detected when no user is present. Bayou does not take the approach of systems that mark conflicting data as unavailable until a person resolves the conflict. Instead, clients can read data at all times, including data whose conflicts have not been fully resolved either because human intervention is needed or because other conflicting updates may be propagating through the system. Bayou provides interfaces that make the state of a replica's data apparent to the application.

The contributions presented in this paper are as follows: we introduce per-update dependency checks and merge procedures as

Preprint of paper to appear in the Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 3-6, 1995, Copper Mountain Resort, Colorado.

a general mechanism for application-specific conflict detection and resolution; we define two states of an update, committed and tentative, which relate to whether or not the conflicts potentially introduced by the update have been ultimately resolved; we present mechanisms for managing these two states of an update both from the perspective of the clients and the storage management requirements of the replicas; we describe how replicas move towards eventual consistency; and, finally, we discuss how security is provided in a system like Bayou.

2. Bayou Applications

The Bayou replicated storage system was designed to support a variety of non-real-time collaborative applications, such as shared calendars, mail and bibliographic databases, program development, and document editing for disconnected workgroups, as well as applications that might be used by individuals at different hosts at different times. To serve as a backdrop for the discussion in following sections, this section presents a quick overview of two applications that have been implemented thus far, a meeting room scheduler and a bibliographic database.

2.1 Meeting room scheduler

Our meeting room scheduling application enables users to reserve meeting rooms. At most one person (or group) can reserve the room for any given period of time. This meeting room scheduling program is intended for use after a group of people have already decided that they want to meet in a certain room and have determined a set of acceptable times for the meeting. It does not help them to determine a mutually agreeable place and time for the meeting, it only allows them to reserve the room. Thus, it is a much simpler application than one of general meeting scheduling.

Users interact with a graphical interface for the schedule of a room that indicates which times are already reserved, much like the display of a typical calendar manager. The meeting room scheduling program periodically re-reads the room schedule and refreshes the user's display. This refresh process enables the user to observe new entries added by other users. The user's display might be out-of-date with respect to the confirmed reservations of the room, for example when it is showing a local copy of the room schedule on a disconnected laptop.

Users reserve a time slot simply by selecting a free time period and filling in a form describing the meeting that is being scheduled. Because the user's display might be out-of-date, there is a chance that the user could try to schedule a meeting at a time that was already reserved by someone else. To account for this possibility, users can select several acceptable meeting times rather than just one. At most one of the requested times will eventually be reserved.

A user's reservation, rather than being immediately confirmed (or rejected), may remain "tentative" for awhile. While tentative, a meeting may be rescheduled as other interfering reservations become known. Tentative reservations are indicated as such on the display (by showing them grayed). The "outdatedness" of a calendar does not prevent it from being useful, but simply increases the likelihood that tentative room reservations will be rescheduled and finally "committed" to less preferred meeting times.

A group of users, although disconnected from the rest of the system, can immediately see each other's tentative room reservations if they are all connected to the same copy of the meeting room schedule. If, instead, users are maintaining private copies on their laptop computers, local communication between the machines will eventually synchronize all copies within the group.

2.2 Bibliographic database

Our second application allows users to cooperatively manage databases of bibliographic entries. Users can add entries to a database as they find papers in the library, in reference lists, via word of mouth, or by other means. A user can freely read and write any copy of the database, such as one that resides on his laptop. For the most part, the database is append-only, though users occasionally update entries to fix mistakes or add personal annotations.

As is common in bibliographic databases, each entry has a unique, human-sensible key that is constructed by appending the year in which the paper was published to the first author's last name and adding a character if necessary to distinguish between multiple papers by the same author in the same year. Thus, the first paper by Jones *et al.* in 1995 might be identified as "Jones95" and subsequent papers as "Jones95b", "Jones95c", and so on.

An entry's key is tentatively assigned when the entry is added. A user must be aware that the assigned keys are only tentative and may change when the entry is "committed." In other words, a user must be aware that other concurrent updaters could be trying to assign the same key to different entries. Only one entry can have the key; the others will be assigned alternative keys by the system. Thus, for example, if the user employs the tentatively assigned key in some fashion, such as embedding it as a citation in a document, then he must also remember later to check that the key assigned when the entry was committed is in fact the expected one.

Because users can access inconsistent database copies, the same bibliographic entry may be concurrently added by different users with different keys. To the extent possible, the system detects duplicates and merges their contents into a single entry with a single key.

Interestingly, this is an application where a user may choose to operate in disconnected mode even if constant connectivity were possible. Consider the case where a user is in a university library looking up some papers. He occasionally types bibliographic references into his laptop or PDA. He may spend hours in the library but only enter a handful of references. He is not likely to want to keep a cellular phone connection open for the duration of his visit. Nor will he want to connect to the university's local wireless network and subject himself to student hackers. He will more likely be content to have his bibliographic entries integrated into his database stored by Bayou upon returning to his home or office.

3. Bayou's Basic System Model

In the Bayou system, each *data collection* is replicated in full at a number of *servers*. Applications running as *clients* interact with the servers through the Bayou application programming interface (API), which is implemented as a client stub bound with the application. This API, as well as the underlying client-server RPC protocol, supports two basic operations: *Read* and *Write*. Read operations permit queries over a data collection, while Write operations can insert, modify, and delete a number of data items in a collection. Figure 1 illustrates these components of the Bayou architecture. Note that a client and a server may be co-resident on a host, as would be typical of a laptop or PDA running in isolation.

Access to one server is sufficient for a client to perform useful work. The client can read the data held by that server and submit Writes to the server. Once a Write is accepted by a server, the client has no further responsibility for that Write. In particular, the client does not wait for the Write to propagate to other servers. In other words, Bayou presents a weakly consistent replication model with a *read-any/write-any* style of access. Weakly consistent replication has been used previously for availability, simplicity and scalability in a variety of systems [3, 7, 10, 12, 15, 19].

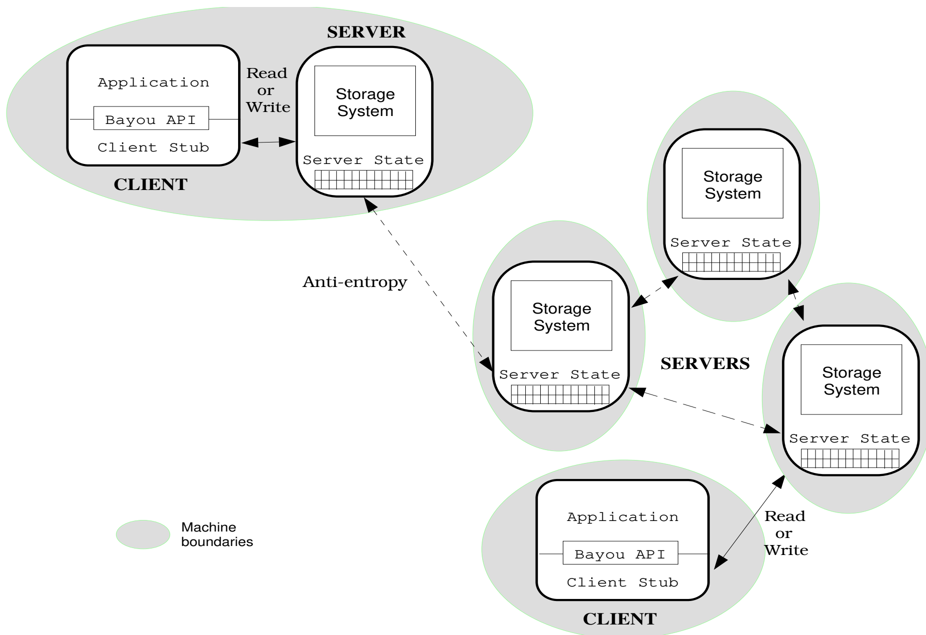

Figure 1. Bayou System Model

While individual Read and Write operations are performed at a single server, clients need not confine themselves to interacting with a single server. Indeed, in a mobile computing environment, switching between servers is often desirable, and Bayou provides *session guarantees* to reduce client-observed inconsistencies when accessing different servers. The description of session guarantees has been presented elsewhere [29].

To support application-specific conflict detection and resolution, Bayou Writes must contain more than a typical file system write or database update. Along with the desired updates, a Bayou Write carries information that lets each server receiving the Write decide if there is a conflict and if so, how to fix it. Each Bayou Write also contains a globally unique *WriteID* assigned by the server that first accepted the Write.

The storage system at each Bayou server conceptually consists of an ordered log of the Writes described above plus the data resulting from the execution of these Writes. Each server performs each Write locally with conflicts detected and resolved as they are encountered during the execution. A server immediately makes the effects of all known Writes available for reading.

In keeping with the goal of requiring as little of the network as possible, Bayou servers propagate Writes among themselves during pair-wise contacts, called *anti-entropy* sessions [7]. The two servers involved in a session exchange Write operations so that when they are finished they agree on the set of Bayou Writes they have seen and the order in which to perform them.

The theory of epidemic algorithms assures that as long as the set of servers is not permanently partitioned each Write will eventually reach all servers [7]. This holds even for communication patterns in which at most one pair of servers is ever connected at once. In the absence of new Writes from clients, all servers will eventually hold the same data. The rate at which servers reach convergence depends on a number of factors including network connectivity, the frequency of anti-entropy, and the policies by which servers select anti-entropy partners. These policies may vary according to the characteristics of the network, the data, and its servers. Developing optimal anti-entropy policies is a research topic in its own right and not further discussed in this paper.

4. Conflict Detection and Resolution

4.1 Accommodating application semantics

Supporting application-specific conflict detection and resolution is a major emphasis in the Bayou design. A basic tenet of our work is that storage systems must provide means for an application to specify its notion of a conflict along with its policy for resolving conflicts. In return, the system implements the mechanisms for reliably detecting conflicts, as specified by the application, and for automatically resolving them when possible. This design goal follows from the observation that different applications have different notions of what it means for two updates to conflict, and that such conflicts cannot always be identified by simply observing conventional reads and writes submitted by the applications.

As an example of application-specific conflicts, consider the meeting room scheduling application discussed in Section 2.1. Observing updates at a coarse granularity, such as the whole-file level, the storage system might detect that two users have concurrently updated different replicas of the meeting room calendar and conclude that their updates conflict. Observing updates at a fine granularity, such as the record level, the system might detect that the two users have added independent records and thereby conclude that their updates do not conflict. Neither of these conclusions are warranted. In fact, for this application, a conflict occurs when two meetings scheduled for the same room overlap in time.

Bibliographic databases provide another example of application-specific conflicts. In this application, two bibliographic entries conflict when either they describe different publications but have been assigned the same key by their submitters or else they describe the same publication and have been assigned distinct keys. Again, this definition of conflicting updates is specific to this application.

The steps taken to resolve conflicting updates once they have been detected may also vary according to the semantics of the application. In the case of the meeting room scheduling application, one or more of a set of conflicting meetings may need to be

```
Bayou_Write (update, dependency_check, mergeproc) {
    IF (DB_Eval (dependency_check.query) <> dependency_check.expected_result)
        resolved_update = Interpret (mergeproc);
    ELSE
        resolved_update = update;
    DB_Apply (resolved_update);
}
```

Figure 2. Processing a Bayou Write Operation

```
Bayou_Write(
    update = {insert, Meetings, 12/18/95, 1:30pm, 60min, "Budget Meeting"},
    dependency_check = {
        query = "SELECT key FROM Meetings WHERE day = 12/18/95
            AND start < 2:30pm AND end > 1:30pm",
        expected_result = EMPTY},
    mergeproc = {
        alternates = {{12/18/95, 3:00pm}, {12/19/95, 9:30am}};
        newupdate = {};
        FOREACH a IN alternates {
            # check if there would be a conflict
            IF (NOT EMPTY (
                SELECT key FROM Meetings WHERE day = a.date
                 AND start < a.time + 60min AND end > a.time))
                    CONTINUE;
            # no conflict, can schedule meeting at that time
            newupdate = {insert, Meetings, a.date, a.time, 60min, "Budget Meeting"};
            BREAK;
        }
        IF (newupdate = {})   # no alternate is acceptable
            newupdate = {insert, ErrorLog, 12/18/95, 1:30pm, 60min, "Budget Meeting"};
        RETURN newupdate;}
)
```

Figure 3. A Bayou Write Operation

moved to a different room or different time. In the bibliographic application, an entry may need to be assigned a different unique key or two entries for the same publication may need to be merged into one.

The Bayou system includes two mechanisms for automatic conflict detection and resolution that are intended to support arbitrary applications: *dependency checks* and *merge procedures*. These mechanisms permit clients to indicate, for each individual Write operation, how the system should detect conflicts involving the Write and what steps should be taken to resolve any detected conflicts based on the semantics of the application. They were designed to be flexible since we expect that applications will differ appreciably in both the procedures used to handle conflicts, and, more generally, in their ability to deal with conflicts.

Techniques for semantic-based conflict detection and resolution have previously been incorporated into some systems to handle special cases such as file directory updates. For example, the Locus [30], Ficus [12], and Coda [17] distributed file systems all include mechanisms for automatically resolving certain classes of conflicting directory operations. More recently, some of these systems have also incorporated support for "resolver" programs that reduce the need for human intervention when resolving other types of file conflicts [18, 26]. Oracle's symmetric replication product also includes the notion of application-selected resolvers for relational databases [8]. Other systems, like Lotus Notes [15], do not provide application-specific mechanisms to handle conflicts, but rather create multiple versions of a document, file, or data object when conflicts arise. As will become apparent from the next couple of sections, Bayou's dependency checks and merge procedures are more general than these previous techniques.

4.2 Dependency checks

Application-specific conflict detection is accomplished in the Bayou system through the use of *dependency checks*. Each Write operation includes a dependency check consisting of an application-supplied query and its expected result. A conflict is detected if the query, when run at a server against its current copy of the data, does not return the expected result. This dependency check is a precondition for performing the update that is included in the Write operation. If the check fails, then the requested update is not performed and the server invokes a procedure to resolve the detected conflict as outlined in Figure 2 and discussed below.

As an example of application-defined conflicts, Figure 3 presents a sample Bayou Write operation that might be submitted by the meeting room scheduling application. This Write attempts to reserve an hour-long time slot. It includes a dependency check with a single query, written in an SQL-like language, that returns information about any previously reserved meetings that overlap with this time slot. It expects the query to return an empty set.

Bayou's dependency checks, like the version vectors and timestamps traditionally used in distributed systems [12, 19, 25, 27], can be used to detect Write-Write conflicts. That is, they can be used to detect when two users update the same data item without one of them first observing the other's update. Such conflicts can be detected by having the dependency check query the current values of any data items being updated and ensure that they have not changed from the values they had at the time the Write was submitted, as is done in Oracle's replicated database [8].

Bayou's dependency checking mechanism is more powerful than the traditional use of version vectors since it can also be used to detect Read-Write conflicts. Specifically, each Write operation can explicitly specify the expected values of any data items on which the update depends, including data items that have been read but are not being updated. Thus, Bayou clients can emulate the optimistic style of concurrency control employed in some distributed database systems [4, 6]. For example, a Write operation that installs a new program binary file might only include a dependency check of the sources, including version stamps, from which it was derived. Since the binary does not depend on its previous value, this need not be included.

Moreover, because dependency queries can read any data in the server's replica, dependency checks can enforce arbitrary, multi-item integrity constraints on the data. For example, suppose a Write transfers \$100 from account A to account B. The application, before issuing the Write, reads the balance of account A and discovers that it currently has \$150. Traditional optimistic concurrency control would check that account A still had \$150 before performing the requested Write operation. The real requirement, however, is that the account have at least \$100, and this can easily be specified in the Write's dependency check. Thus, only if concurrent updates cause the balance in account A to drop below \$100 will a conflict be detected.

4.3 Merge procedures

Once a conflict is detected, a *merge procedure* is run by the Bayou server in an attempt to resolve the conflict. Merge procedures, included with each Write operation, are general programs written in a high-level, interpreted language. They can have embedded data, such as application-specific knowledge related to the update that was being attempted, and can perform arbitrary Reads on the current state of the server's replica. The merge procedure associated with a Write is responsible for resolving any conflicts detected by its dependency check and for producing a revised update to apply. The complete process of detecting a conflict, running a merge procedure, and applying the revised update, shown in Figure 2, is performed atomically at each server as part of executing a Write.

In principle, the algorithm in Figure 2 could be imbedded in each merge procedure, thereby eliminating any special mechanisms for dependency checking. This approach would require servers to create a new merge procedure interpreter to execute each Write, which would be overly expensive. Supporting dependency checks separately allows servers to avoid running the merge procedure in the expected case where the Write does not introduce a conflict.

The meeting room scheduling application provides good examples of conflict resolution procedures that are specific not only to a particular application but also to a particular Write operation. In this application, users, well aware that their reservations may be invalidated by other concurrent users, can specify alternate scheduling choices as part of their original scheduling updates. These alternates are encoded in a merge procedure that attempts to reserve one of the alternate meeting times if the original time is found to be in conflict with some other previously scheduled meeting. An example of such a merge procedure is illustrated in Figure 3. A different merge procedure altogether could search for the next available time slot to schedule the meeting, which is an option a user might choose if any time would be satisfactory.

In practice, Bayou merge procedures are written by application programmers in the form of templates that are instantiated with the appropriate details filled in for each Write. The users of applications do not have to know about merge procedures, and therefore about the internal workings of the applications they use, except when automatic conflict resolution cannot be done.

In the case where automatic resolution is not possible, the merge procedure will still run to completion, but is expected to produce a revised update that logs the detected conflict in some fashion that will enable a person to resolve the conflict later. To enable manual resolution, perhaps using an interactive merge tool [22], the conflicting updates must be presented to a user in a manner that allows him to understand what has happened. By convention, most Bayou data collections include an error log for unresolvable conflicts. Such conventions, however, are outside the domain of the Bayou storage system and may vary according to the application.

In contrast to systems like Coda [18] or Ficus [26] that lock individual files or complete file volumes when conflicts have been detected but not yet resolved, Bayou allows replicas to always remain accessible. This permits clients to continue to Read previously written data and to continue to issue new Writes. In the meeting room scheduling application, for example, a user who only cares about Monday meetings need not concern himself with scheduling conflicts on Wednesday. Of course, the potential drawback of this approach is that newly issued Writes may depend on data that is in conflict and may lead to cascaded conflict resolution.

Bayou's merge procedures resemble the previously mentioned resolver programs, for which support has been added to a number of replicated file systems [18, 26]. In these systems, a file-type-specific resolver program is run when a version vector mismatch is detected for a file. This program is presented with both the current and proposed file contents and it can do whatever it wishes in order to resolve the detected conflict. An example is a resolver program for a binary file that checks to see if it can find a specification for how to derive the file from its sources, such as a Unix makefile, and then recompiles the program in order to obtain a new, "resolved" value for the file. Merge procedures are more general since they can vary for individual Write operations rather than being associated with the type of the updated data, as illustrated above for the meeting room scheduling application.

5. Replica Consistency

While the replicas held by two servers at any time may vary in their contents because they have received and processed different Writes, a fundamental property of the Bayou design is that all servers move towards *eventual consistency.* That is, the Bayou system guarantees that all servers *eventually* receive all Writes via the pair-wise anti-entropy process and that two servers holding the same set of Writes will have the *same* data contents. However, it cannot enforce strict bounds on Write propagation delays since these depend on network connectivity factors that are outside of Bayou's control.

Two important features of the Bayou system design allows servers to achieve eventual consistency. First, Writes are performed in the same, well-defined order at all servers. Second, the conflict detection and merge procedures are deterministic so that servers resolve the same conflicts in the same manner.

In theory, the execution history at individual servers could vary as long as their execution was *equivalent to* some global Write ordering. For example, Writes known to be commutative could be performed in any order. In practice, because Bayou's Write operations include arbitrary merge procedures, it is effectively impossible either to determine whether two Writes commute or to transform two Writes so they can be reordered as has been suggested for some systems [9].

When a Write is accepted by a Bayou server from a client, it is initially deemed *tentative*. Tentative Writes are ordered according to timestamps assigned to them by their accepting servers. Eventually, each Write is *committed*, by a process described in the next section. Committed Writes are ordered according to the times at which they commit and before any tentative Writes.

The only requirement placed on timestamps for tentative Writes is that they be monotonically increasing at each server so that the pair <timestamp, ID of server that assigned it> produce a total order on Write operations. There is no requirement that servers have synchronized clocks, which is crucial since trying to ensure clock synchronization across portable computers is problematic. However, keeping servers' clocks reasonably close is desirable so that the induced Write order is consistent with a user's perception of the order in which Writes are submitted. Bayou servers maintain logical clocks [20] to timestamp new Writes. A server's logical clock is generally synchronized with its real-time system clock, but, to preserve the causal ordering of Write operations, the server may need to advance its logical clock when Writes are received during anti-entropy.

Enforcing a global order on tentative, as well as committed, Writes ensures that an isolated cluster of servers will come to agreement on the tentative resolution of any conflicts that they encounter. While this is not strictly necessary since clients must be prepared to deal with temporarily inconsistent servers in any case, we believe it desirable to provide as much internal consistency as possible. Moreover, clients can expect that the tentative resolution of conflicts within their cluster will correspond to their eventual permanent resolution, provided that no further conflicts are introduced outside the cluster.

Because servers may receive Writes from clients and from other servers in an order that differs from the required execution order, and because servers immediately apply all known Writes to their replicas, servers must be able to undo the effects of some previous tentative execution of a Write operation and reapply it in a different order. Interestingly, the number of times that a given Write operation is re-executed depends only on the order in which Writes arrive via anti-entropy and not on the likelihood of conflicts involving the Write.

Conceptually, each server maintains a log of all Write operations that it has received, sorted by their committed or tentative timestamps, with committed Writes at the head of the log. The server's current data contents are generated by executing all of the Writes in the given order. Techniques for pruning a server's Write log and for efficiently maintaining the corresponding data contents by undoing and redoing Write operations are given in Section 7.

Bayou guarantees that merge procedures, which execute independently at each server, produce consistent updates by restricting them to depend only on the server's current data contents and on any data supplied by the merge procedure itself. In particular, a merge procedure cannot access time-varying or server-specific "environment" information such as the current system clock or server's name. Moreover, merge procedures that fail due to exceeding their limits on resource usage must fail deterministically. This means that all servers must place uniform bounds on the CPU and memory resources allocated to a merge procedure and must consistently enforce these bounds during execution. Once these conditions are met, two servers that start with identical replicas will end up with identical replicas after executing a Write.

6. Write Stability and Commitment

A Write is said to be *stable* at a server when it has been executed for the last time by that server. Recall that as servers learn of new updates by performing anti-entropy with other servers, the effects of previously executed Write operations may need to be undone and the Writes re-executed. Thus, a given Write operation may be executed several times at a server and may produce different results depending on the execution history of the server. A Write operation becomes stable when the set of Writes that precede it in the server's Write log is fixed. This means that the server has already received and executed any Writes that could possibly be ordered before the given Write. Bayou's notion of stability is similar to that in ordered multicast protocols, such as those provided in the ISIS toolkit [2].

In many cases, an application can be designed with a notion of "confirmation" or "commitment" that corresponds to the Bayou notion of stability. As an example, in the Bayou meeting room scheduling application, two users may try to schedule separate meetings for the same time in the same room. Only when one of the users discovers that his Write has become stable and his schedule still shows that he has reserved the room for the desired time, can he be sure that his tentative reservation has been confirmed.

Since clients may want to know when a Write has stabilized, the Bayou API provides means for inquiring about the stability of a specific Write. Given a Write's unique identifier, a client can ask a server whether the given Write is stable at the server. The answer may vary, of course, depending on which server is contacted. Bayou also provides support for clients that may choose to access only stable data.

How does a server determine whether a Write is stable? One approach would be to have each server include in the information passed during anti-entropy not only any Writes that have been accepted by this server but also the current value of the clock that it uses to timestamp new Writes. With suitable assumptions about the propagation order of Writes, a server could then determine that a Write is stable when it has a lower timestamp than all servers' clocks. The main drawback of this approach is that a server that remains disconnected can prevent Writes from stabilizing, which could cause a large number of Writes to be rolled back when the server reconnects.

To speed up the rate at which updates stabilize in an environment where communication with some servers may not be possible for extended periods of time, the Bayou system uses a *commit* procedure. That is, a Write becomes stable when it is explicitly committed, and, in fact, we generally use the terms "stable" and "committed" interchangeably in the Bayou system. Committed Writes, in commit order, are placed ahead of any tentative Writes in each server's Write log. This, along with Bayou's anti-entropy protocol ensuring that servers learn of committed Writes in the order that they were committed, provides stability.

In the Bayou system, we use a *primary commit* scheme [28]. That is, one server designated as the *primary* takes responsibility for committing updates. Knowledge of which Writes have committed and in which order they were committed then propagates to other servers during anti-entropy. In all other respects, the primary behaves exactly like any other server. Each replicated data collection can have a different server designated as its primary.

Any commit protocol that prevents different groups of servers from committing updates in different orders would meet Bayou's needs. In our anticipated weak connectivity environment, using a primary to commit data is more attractive than the standard two-phase commit protocol since it alleviates the need to gather a majority quorum of servers. Consider the case of data that is replicated among laptops that are mostly disconnected. Requiring a majority of these laptops to be in communication with each other

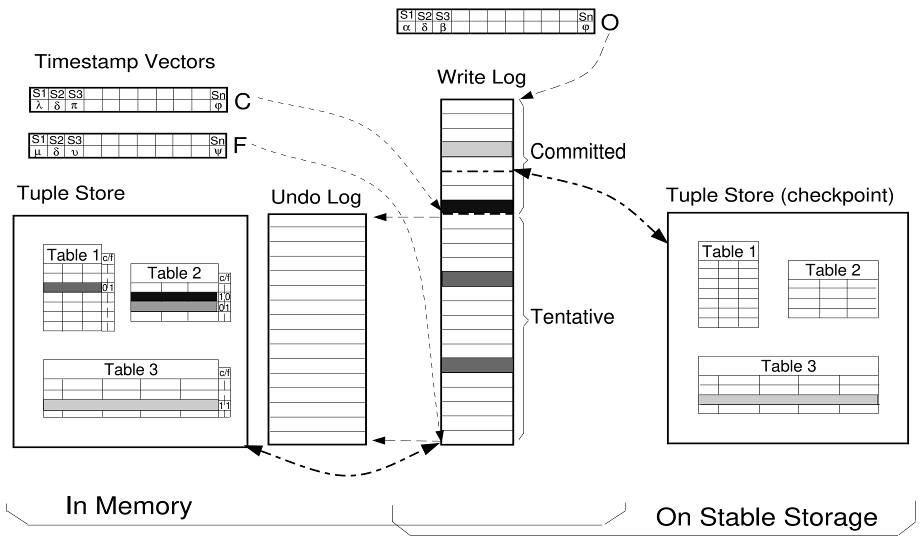

Figure 4. Bayou Database Organization

at the same time in order to commit updates would be unreasonable.

The primary commit approach also enables updates to commit on a disconnected laptop that acts as the primary server. For example, suppose a user keeps the primary copy of his calendar with him on his laptop and allows others, such as a spouse or secretary, to keep secondary, mostly read-only copies. In this case, the user's updates to his own calendar commit immediately. This example illustrates how one might choose the primary to coincide with the locus of update activity, thereby maximizing the rate at which Writes get committed.

Unlike other distributed storage systems in which the ability to commit data is of primary importance, the Bayou design readily accommodates the temporary unavailability of the primary. The inability of a client to communicate with the primary server, for instance if the primary crashes or is disconnected, does not prevent it from performing useful Read and Write operations. Writes accepted by other servers simply remain tentative until they eventually reach the primary.

Bayou tries to arrange, but cannot ensure, that the order in which Writes are committed is consistent with the tentative order indicated by their timestamps. Writes from a given server are committed in timestamp order. Writes from different servers, however, may commit in a different order based on when the servers perform anti-entropy with the primary and with each other. Writes held on a disconnected non-primary server, for instance, will commit only after the server reconnects to the rest of the system and could be committed after Writes with later timestamps.

7. Storage System Implementation Issues

The Bayou design places several demands on the underlying storage system used by each server including the need for space-efficient Write logging, efficient undo/redo of Write operations, separate views of committed and tentative data, and support for server-to-server anti-entropy. We implemented a storage system tailored to these special needs.

Our implementation is factored into three main components as shown in Figure 4: the *Write Log,* the *Tuple Store,* and the *Undo Log*. The Write Log contains Writes that have been received by a Bayou server, sorted by their global committed or tentative order. The server's Tuple Store is a database that is obtained by executing the Writes in order and is used to process Read requests. The Undo Log facilitates rolling back tentative Writes that have been applied to the Tuple Store so that they can be re-executed in a different order, such as when a newly received Write gets inserted into the middle of the Write Log or when existing Writes get reordered through the commit process.

The Write Log conceptually contains all Writes ever received by the server, as discussed in Section 5. In practice, a server can discard a Write from the Write Log once it becomes stable, since by definition the server will never need to rollback and re-execute a stable Write. Bayou servers do, in fact, hold onto a few recently committed Writes to facilitate incremental anti-entropy, the details of which are beyond the scope of this paper. Thus, the Write Log is actually an ordered set of Writes containing a tail of the committed Writes and all tentative Writes known to the server.

Each server must keep track of which Writes it has received but are no longer explicitly held in its Write Log. This is to ensure that the server does not re-accept the same Writes from another server during anti-entropy. Each server maintains a timestamp vector, called the "O vector", to indicate in a compact way the "omitted" prefix of committed Writes. This O vector records, for each server, the timestamp of the latest Write from the given server that has been discarded. A single timestamp vector can precisely characterize the set of discarded Writes because: (1) servers discard a prefix of their Write Log, and (2) Writes that originate from any given server propagate and get committed in timestamp order.

The Tuple Store we implemented is an in-memory relational database, providing query processing in a subset of SQL, local transaction support, and some integrity constraints. Requiring a database to fit in virtual memory is, admittedly, a practical limitation in our current implementation, but is not intrinsic to the overall Bayou design. The Tuple Store, and its associated language for specifying queries and updates, is the principal place in the Bayou architecture where the issue of data model arises. We chose the relational model for our initial prototype because of its power and flexibility. It naturally supports fine-grain access to specific fields of tuples as well as queries and updates to all tuples in the database.

A unique aspect of the Tuple Store is that it must support the two distinct views of a Bayou database that are of interest to clients: *committed* and *full*. When a Write is tentative, its effect appears in the full view but not in the committed view. Once the Write has been committed, its effect appears in both views. A ten-

```
Receive_Writes (writeset, received_from) {
    IF (received_from = CLIENT) {
        # Received one write from the client, insert at end of WriteLog
        # first increment the server's timestamp
        logicalclock = MAX(systemclock, logicalclock + 1);
        write = First(writeset);
        write.WID = {logicalclock, myServerID};
        write.state = TENTATIVE;
        WriteLog_Append(write);
        Bayou_Write(write.update, write.dependency_check, write.mergeproc);
    } ELSE {
        # Set of writes received from another server during anti-entropy,
        # therefore writeset is ordered
        write = First(writeset);
        insertionPoint = WriteLog_IdentifyInsertionPoint(write.WID);
        TupleStore_RollbackTo(insertionPoint);
        WriteLog_Insert(writeset);
        # Now roll forward
        FOREACH write IN WriteLog AFTER insertionPoint DO
            Bayou_Write(write.update, write.dependency_check, write.mergeproc);
        # Maintain the logical clocks of servers close
        write = Last(writeset);
        logicalclock = MAX(logicalclock, write.WID.timestamp);
    }
}
```

Figure 5. Applying Sets of Bayou Writes to the Database

tative deletion may result in a tuple that appears in the committed view but not in the full view. For many servers, certainly those that communicate regularly with the primary, the committed and full views will be nearly identical. However, neither view is a subset of the other.

Our Tuple Store maintains the union of the two views. Each tuple is tagged with a 2-bit characteristic vector identifying the set of views that contain it. The bits of all tuples affected by a Write get set when the Write is applied to the Tuple Store. Therefore, re-executing a Write when it gets committed is necessary so that all corresponding committed bits get set appropriately. Our query processor respects and propagates these bits, so that in the result of a query each tuple is tagged with the views for which that tuple would be produced if the identical query were run conventionally. Propagating these bits through a relational algebra query is straightforward. Assuming the tentative and committed views are nearly identical, this technique reduces the space occupied by the Tuple Store, compared to maintaining two separate full and committed databases, by nearly a factor of two without substantially increasing the query processing cost. In addition, our query processor can easily guarantee that identical tuples occurring in the two views of a query result will always be merged and delivered as a single tuple with both bits in the characteristic vector set. This makes it convenient for clients to base decisions on the *difference* between the two views without having to merge the results of independent queries.

To support anti-entropy efficiently, the running state of each server also includes two timestamp vectors that represent the committed and full views. The “C vector” characterizes the state of the Tuple Store after executing the last committed Write in the Write Log while the “F vector” characterizes the state after executing the last tentative Write in the Write Log, that is, the current Tuple Store. These timestamp vectors are *not* used for conflict detection; they simply enable server pairs to identify precisely the sets of Writes that need to be exchanged during anti-entropy.

The Undo Log permits a server to undo any effects on the Tuple Store of Writes performed after a given position in the Write Log. As each new Write is received via anti-entropy, a server inserts it into its Write Log. Newly committed Writes are inserted immediately following the current set of committed Writes known to the server, which may in turn require that some of these Writes be removed from their previous tentative positions in the Write Log. After all the Writes have been received, the server uses its Undo Log to roll back its Tuple Store to a state corresponding to the position where the first newly received Write was inserted. It then enumerates and executes all following Writes from the Write Log, bringing its Tuple Store and Undo Log up-to-date. This procedure is illustrated in Figure 5.

For crash recovery purposes, both the full Write Log and a checkpoint of the Tuple Store are maintained in stable storage, while for performance the Write Log and the current Tuple Store are maintained in memory as shown in Figure 4. The Undo Log is maintained only in memory. The stable checkpoint of the Tuple Store reflects only a prefix of the committed Writes. This checkpoint must contain the effects of any Writes that have been truncated from the Write Log. At all times, a valid Tuple Store can be recovered by reading this checkpoint and applying a suffix of the Write Log to it. Thus, to make the database recoverable, Bayou stably records the unique identifier of the last Write reflected in the Tuple Store checkpoint (making it possible to identify the correct suffix of the Write Log) and makes the Write Log itself recoverable using conventional techniques for logging high-level changes to the Write Log.

8. Access Control

Providing access control and authentication in Bayou posed interesting challenges because of our minimal connectivity assumptions. In particular, the design cannot rely on an online, trusted authentication server [23] to mediate the establishment of secure channels between a client and server or between two Bayou servers. As an example, suppose two users holding Bayou replicas on their portable computers are in a meeting together. Before performing anti-entropy, each of the two mutually suspicious servers must verify that the other is authorized to manage the data. Similarly, if one machine simply wants to act as a client for the data stored on the other, it will want to make sure that the server is legitimate and then must prove that it is authorized to access the data.

The access control model currently implemented in the Bayou system provides authorization at the granularity of a whole data collection, which is the unit of replication. A user may be granted Read and Write privileges to a data collection. A user may also be granted "Server" privileges to maintain a replica of the data on his workstation or portable computer, that is, to run a server for the data collection. Enabling servers to run on mobile platforms radically departs from the notion of physically protected servers.

Mutual authentication and access control in Bayou is based on public-key cryptography. Every user possesses a public/private key pair and a set of digitally signed *access control certificates* granting him access to various data collections. Client applications and Bayou servers operate on behalf of users and obtain the key pair and access control certificates from the corresponding user at start-up time. Currently, we use a single trusted signing authority with a well-known public key to sign all access-granting certificates, though moving to a hierarchy or web of signing authorities would not be difficult.

Bayou uses three types of certificates to grant, delegate and revoke access to a data collection:

- AC[PU, P, D] - certificate that grants privilege P (one of Read, Write, or Server) on data collection D to the user whose public key is PU. AC certificates are signed by the well-known signing authority.
- D[PU, C, PY] - certificate signed by the user whose public key is PY to delegate his privileges encoded in certificate C to another user whose public key is PU.
- R[C, PY] - certificate signed by the user whose public key is PY to revoke some user's privileges encoded in certificate C; the user whose public key is PY must also have originally signed certificate C.

Revocation certificates are stored by Writes to, and hence propagated with, the data collections to which they apply. Certificates that revoke server privileges may also be kept by client users to ensure protection against malicious servers. Users maintain a *certificate purse*, which applications running under their identity can both read and append to.

For a server to determine whether a client has some privilege for the server's data, the server first authenticates the client's identity using a challenge/response protocol. The client also hands the server a certificate that asserts the privilege in question. The server must verify that the certificate is legitimate, that the certificate and any enclosed certificates for a delegation have not been revoked to the server's knowledge, and that it grants the necessary access rights. Server-to-client and server-to-server authentication and access control checking is done in a similar fashion. The establishment of mutual trust between a server and a client is performed at the beginning of a *secure session* and covers all Read or Write operations performed as part of that session. A server will preempt the session if it is notified of a revocation that affects a certificate associated with the session.

For Write operations, the submitter's access rights are checked once by the accepting server, and then again at the primary when the Write is committed. Servers, other than the primary, when receiving a Write during anti-entropy trust that the accepting server has correctly checked the user's privileges and rejected Writes with unsuitable access rights. This level of trust is reasonable since a server ensures that any server with which it performs anti-entropy is authorized to hold a replica of the data collection.

Having access controls checked for a second time at the primary server ensures that revocations of Write privileges can be applied at the primary and guarantees that any "bad" Write attempting to commit after such a revocation will be rejected. In particular, revocation of Write access for a malicious user can be enforced without having to ensure that every server to which such a user could connect has been notified of the revocation.

Even though a Write's merge procedure may perform different Read operations on the data and perform different updates when it is executed at different times, checking access control once is sufficient because of the whole-data-collection access control model. More fine-grained access control would require careful design modifications.

9. Status and Experience

The implementation of the Bayou system has two distinguishable components: the client stub and the server. The client stub is a runtime library linked into applications that use Bayou for storage management. It provides mechanisms for server location, session guarantees, secure sessions, Read and Write operations, and miscellaneous utilities. The server implements the Bayou storage management including the mechanisms for conflict detection and resolution, server to server communication, and persistent database management. Bayou's implementation is Posix compliant and developed in ANSI C so that the same sources run on Intel-based laptops with Linux and on our regular development platform of Sun SPARCstations with SunOS.

In the current implementation, ILU [14], a language-independent RPC package developed at Xerox PARC, is used for communication between Bayou clients and servers, as well as between servers. Server location, by both clients and other servers, uses a simple decentralized registration and lookup service for key-value pairs that are made visible across a network via multicast. Bayou merge procedures are Tcl programs [24] that are run in a Tcl interpreter modified to enforce the limits described in Section 5. We foresee that these components may change as the system evolves.

The two running applications have demonstrated how to use Bayou's conflict detection and resolution mechanisms effectively. Interestingly, one of the lessons we learned immediately from these applications was that the Bayou server had to supply a per-database library mechanism for Tcl code invoked by the merge procedures. Otherwise, Writes are bloated by the large amount of repeated code in their merge procedures. For both the meeting room scheduler and the shared bibliographic database manager only two of roughly 100 lines of Tcl in the original merge procedures changed from one Write to another.

The performance of Bayou depends on several factors, such as the schema of the data being stored, the amount of data stored at a server, the location of clients and servers, and the platforms on which the components are running. This section shows how Bayou performs for a particular instance of the system: a server and client for the bibliographic database described in Section 2. The database is composed of a single table containing 1550 tuples, obtained from a bibtex source [21]. Each tuple was inserted into the database with a single Bayou Write operation. Results are presented for five different configurations of the database characterized by the number of Writes that are tentative. For each configuration we measured storage requirements and the execution times for three operations in the system: undoing and redoing the effect of all tentative Writes, executing a client Read operation against the database, and adding a new Write to the database.

Table 1: Size of Bayou Storage System for the Bibliographic Database with 1550 Entries
(sizes in Kilobytes)

Number of Tentative Writes	0 (none)	50	100	500	1550 (all)
Write Log	9	129	259	1302	4028
Tuple Store Ckpt	396	384	371	269	1
Total	**405**	**513**	**630**	**1571**	**4029**
Factor to 368K bibtex source	1.1	1.39	1.71	4.27	10.95

Table 2: Performance of the Bayou Storage System for Operations on Tentative Writes in the Write Log
(times in milliseconds with standard deviations in parentheses)

Tentative Writes	0	50		100		500		1550	
	Server running on a Sun SPARC/20 with Sunos								
Undo all	0	31	(6)	70	(20)	330	(155)	866	(195)
(avg. per Write)		.62		.7		.66		.56	
Redo all	0	237	(85)	611	(302)	2796	(830)	7838	(1094)
(avg. per Write)		4.74		6.11		5.59		5.05	
	Server running on a Gateway Liberty Laptop with Linux								
Undo all	0	47	(3)	104	(7)	482	(15)	1288	(62)
(avg. per Write)		.94		1.04		.96		.83	
Redo all	0	302	(91)	705	(134)	3504	(264)	9920	(294)
(avg. per Write)		6.04		7.05		7.01		6.4	

Table 3: Performance of the Bayou Client Operations
(times in milliseconds with standard deviations in parentheses)

Server Client	Sun SPARC/20 same as server		Gateway Liberty same as server		Sun SPARC/20 Gateway Liberty	
Read: 1 tuple	27	(19)	38	(5)	23	(4)
100 tuples	206	(20)	358	(28)	244	(10)
Write: no conflict	159	(32)	212	(29)	177	(22)
with conflict	207	(37)	372	(17)	223	(40)

Table 1 shows that the size of a tentative Write for this database is about 10 times that of a committed Write. Over half of a tentative Write's size is taken by the access control certificate required for security. The server's storage requirements decrease significantly as data gets committed. When most of the Writes in the database are committed, its size is almost identical to that of the bibtex file from which the data was obtained.

Table 2 illustrates the execution times for a Bayou server to undo and then redo all Writes that are tentative in each configuration of the bibliographic database. Each result corresponds to the average over 100 executions of the undo/redo operations. The cost incurred by the server is a function of the number of Writes being undone and redone. While in general the size of the Tuple Store may affect the performance of executing a Write, the cost of redoing each tentative Write for this database is close to constant because dependency checks are selections on the database's primary key index, and are therefore independent of the Tuple Store size. The standard deviations on the Sun tend to be higher than those for the laptop since the Sun workstation was running a much higher workload of other applications than the laptop.

Table 3 shows the performance of client/server interactions for the bibliographic database. Measurements were taken for three computing platform combinations: a Bayou server and bibliographic database client running on the same Sun SPARCstation/20, both server and client running on the same Gateway Liberty laptop, and, finally, the server running on the Sun SPARCstation/20 and the client running on the Gateway Liberty. The numbers for both Reads and Writes include the costs of session guarantee management, the RPC proper, and a database query, which for Writes is part of the dependency check. Additionally, Writes require two file system synchronization operations and, in case of conflict, the execution of the merge procedure, which runs another database query. For Writes involving conflicts, the bibliographic entry key presented in each Write is not unique and, hence, must be reassigned in the merge procedure as discussed in Section 2.2; the key presented is changed after each set of five Write operations. Because Reads operate on the in-memory Tuple Store, running selections on the primary key, and Writes are appended to the Write Log, execution times across the different database configurations vary little. Hence we present the combined average of the 500 executions of each operation over all configurations.

10. Conclusions

Bayou is a storage infrastructure for mobile applications that relies only on weak connectivity assumptions. To cope with arbitrary network partitions, the system is built around pair-wise client-server and server-server communications. To provide high availability, Bayou employs weakly consistent replication where clients are able to connect to any available server to perform Reads and Writes. Support for automatic conflict detection and resolution enables applications to deal with concurrent updates effectively. The system guarantees eventual consistency by ensuring that all updates eventually propagate to all servers, that servers perform updates in a global order, and that any update conflicts are resolved in a consistent manner at all servers.

Bayou's management of update conflicts differs significantly from previous replicated systems, including file systems like Coda [18] and Ficus [26] as well as Oracle's recent commercial database offering [8], in the following main areas:

Non-transparency. Previous systems have tried to support existing file and database applications by detecting and resolving conflicts without the applications' knowledge. In contrast, Bayou adopts the philosophy that applications must be aware of and integrally involved in conflict detection and resolution. Bayou applications can take advantage of the semantics of their data to minimize false conflict detections and maximize the ability to resolve detected conflicts automatically.

Application-specific conflict detection. File systems that rely on version vectors and database systems that employ optimistic concurrency control detect update conflicts by observing clients' Reads and Writes. Using application-provided rules for detecting conflicts, called dependency checks, Bayou can detect a wider class of conflicts, particularly those that depend on application semantics.

Per-write conflict resolvers. Whereas Coda, Ficus, and Oracle all permit clients to write custom procedures to resolve conflicts, these resolvers are stored within the system and invoked based on the type of the file or data in conflict. In Bayou, each Write operation includes, in addition to the desired update and dependency check, a merge procedure that gets executed if the Write is determined to have caused a conflict. The power of this approach is demonstrated in applications, such as Bayou's meeting room scheduler, where the merge procedure varies for each Write.

Partial and multi-object updates. Bayou's Write operations can atomically perform insertions, partial modifications, and deletions to one or more data objects. This means that, unlike systems with a whole-file update model where storing the most recent data contents is sufficient, Bayou servers must apply every Write operation. Techniques have been devised in Bayou for propagating, ordering, and undoing/redoing Write operations to ensure eventual consistency for arbitrary updates and conflict resolution procedures. These techniques are needed not only for relational database models, as in the current Bayou system, but also for file systems supporting record-level updates and multi-file atomic transactions.

Tentative and stable resolutions. The Bayou system is novel in maintaining both full and committed views of the data while permitting clients to read either. Both clients and servers in Bayou want to know when any conflicts involving a Write have been fully resolved. Committing a Write ensures that its outcome is stable, including the resolution of conflicts involving the Write. The rate at which Writes stabilize is independent of the probability of conflict. In keeping with the goal of minimal connectivity requirements, Bayou commits Writes using a primary server.

Security. Bayou executes each merge procedure within a secure environment in which the only allowable external actions are reading and writing data using the access credentials of the user who submitted the conflicting Write. Public-key, digitally-signed certificates permit authentication and access control outside the presence of an authentication server or user.

Applications that can best utilize Bayou's replication scheme are those for which reading weakly consistent, tentative data is acceptable and for which the chance of update conflicts is low or the success of automatic resolution is high. Provided that the penalty for conflict is not excessive, humans would rather deal with the occasional unresolvable conflict than incur the adverse impact on availability inherent in systems that avoid conflicts altogether, such as those based on pessimistic locking. A number of shared databases, such as phone books and bulletin boards, meet these characteristics, as do many asynchronous collaborative applications [22].

We have built an initial version of the Bayou system and our measurements indicate that its performance and overhead are acceptable. In particular, running Bayou servers and applications on today's laptop computers is reasonable. Our measurements also confirm that much of the extra overhead imposed by Bayou's heavier-weight Write operations is present only so long as a Write operation is tentative. Committed data is no more expensive than in other, simpler storage systems. We are also building a number of applications on top of Bayou and experimenting with them to gain better insights into their needs.

Issues we are planning to explore further in the context of Bayou include partial replication, policies for choosing servers for anti-entropy, building servers with conventional database managers, alternate data models, and finer grain access control. Our current focus is on supporting partial replicas that contain subsets of a data collection, which is important for some laptop-based applications and raises a number of difficult problems ranging from characterizing a partial replica to resolving conflicts in a consistent manner across partial replicas. The next steps in the implementation will include the development of other applications, such as an e-mail reader, porting refdbms [11], a widely used shared bibliographic database manager, to run on the Bayou storage system, and experimenting with wireless connectivity for servers and clients running on a laptop.

11. Acknowledgments

The Bayou design has benefitted from discussions with a number of colleagues, including our fellow PARC researchers and Tom Anderson, Mary Baker, Brian Bershad, Hector Garcia-Molina, and Terri Watson. We especially thank Brent Welch for his technical contributions in the early stages of the Bayou project. Atul Adya and Xinhua Zhou helped implement the first Bayou applications, from which we learned a tremendous amount. Surendar Chandra contributed significantly to making the network environment on our laptops work. Sue Owicki helped guide the final revisions to this paper. Mark Weiser and Craig Mudge, as managers of the Computer Science Lab, have been supportive throughout.

12. References

[1] R. Alonso and H. F. Korth. Database system issues in nomadic computing. *Proceedings ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993, pages 388-392.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight, causal and atomic group multicast. *ACM Transactions on Computer Systems* 9(3):272-314, August 1991.

[3] A. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25(4):260-274, April 1982.

[4] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems* 16(4):703-746, December 1991.

[5] B. A. Coan, B. M. Oki, and E. K. Kolodner. Limitations on database availability when networks partition. *Proceedings Fifth ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, Canada, August 1986, pages 187-194.

[6] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: A survey. *ACM Computing Surveys* 17(3):341-370, September 1985.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Proceedings Sixth Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, August 1987, pages 1-12.

[8] A. Downing. Conflict resolution in symmetric replication. *Proceedings European Oracle User Group Conference*, Florence, Italy, April 1995, pages 167-175.

[9] C. Ellis and S. Gibbs. Concurrency control in groupware systems. *Proceedings ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989, pages 399-407.

[10] R. A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems* 5(4):379-405, Fall 1992.

[11] R. Golding, D. Long, and J. Wilkes. The refdbms distributed bibliographic database system. *Proceedings Winter USENIX Conference,* San Francisco, California, January 1994, pages 47-62.

[12] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, Jr., G.J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *Proceedings Summer USENIX Conference*, June 1990, pages 63-71.

[13] T. Imielinski and B. R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM* 37(10):18-28, October 1994.

[14] B. Janssen and M. Spreitzer. *Inter-Language Unification - ILU*. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[15] L. Kalwell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Groupware: Software for Computer-Supported Cooperative Work*, edited by D. Marca and G. Bock, IEEE Computer Society Press, 1992, pages 226-235.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* 10(1): 3-25, February 1992.

[17] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. *Proceedings Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[18] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. *Proceedings USENIX Technical Conference*, New Orleans, Louisiana, January 1995, pages 95-106.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10(4):360-391, November 1992.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558-565, July 1978.

[21] L. Lamport. *LaTeX - a document preparation system*. Addison-Wesley Publishing Company, 1986.

[22] J. P. Munson and P. Dewan. A flexible object merging framework. *Proceedings ACM Conference on Computer Supported Cooperative Work (CSCW)*, Chapel Hill, North Carolina, October 1994, pages 231-242.

[23] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM* 21(12): 993-999, December 1978.

[24] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[25] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* SE-9(3):240-246, May 1983.

[26] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. *Proceedings Summer USENIX Conference*, June 1994, pages 183-195.

[27] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39(4):447-459, April 1990.

[28] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering* SE-5(3):188-194, May 1979.

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

[30] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. Proceedings Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983, pages 49-70.