The Bayou Architecture: Support for Data Sharing among Mobile Users

Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, Brent Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.
contact: terry@parc.xerox.com

Abstract

The Bayou System is a platform of replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications. This paper presents the preliminary system architecture along with the design goals that influenced it. We take a fresh, bottom-up and critical look at the requirements of mobile computing applications and carefully pull together both new and existing techniques into an overall architecture that meets these requirements. Our emphasis is on supporting application-specific conflict detection and resolution and on providing application-controlled inconsistency.

1. Introduction

The Bayou project at Xerox PARC has been designing a system to support data sharing among mobile users. The system is intended to run in a mobile computing environment that includes portable machines with less than ideal network connectivity. In particular, a user's computer may have a wireless communication device, such as a cell modem or packet radio transceiver relying on a network infrastructure that is not universally available and perhaps unreasonably expensive. It may use short-range line-of-sight communication, such as the infrared "beaming" ports available on some commercial personal digital assistants (PDAs). Alternatively, the computer may have a conventional modem requiring it to be physically connected to a phone line when sending and receiving data or may only be able to communicate with the rest of the system when inserted in a docking station. Finally, its only communication device may be a diskette that is transported between machines by humans. The main characteristic of these communication capabilities is that a mobile computer may experience extended and sometimes involuntary disconnection from many or all of the other devices with which it wants to share data.

We believe that mobile users want to share their appointment calendars, bibliographic databases, meeting notes, evolving design documents, news bulletin boards, and other types of data in spite of their intermittent network connectivity. The focus of the Bayou project has been on exploring mechanisms that let mobile clients actively read and write shared data. Even though the system must cope with both voluntary and involuntary communication outages, it should look to users, to the extent possible, like a centralized, highly-available database service. This paper presents detailed goals for the overall system architecture and discusses the design decisions that we made to meet these goals.

2. Architectural design decisions

Goal: Support for portable computers with limited resources.

Design: A flexible client-server architecture.

Many of the devices that we envision being commonly used, such as PDAs and the ParcTab developed within our lab [24], have insufficient storage for holding copies of all, or perhaps any, of the data that their users want to access. For this reason, our architecture is based on a division of functionality between *servers*, which store data, and *clients*, which read and write data managed by servers. A server is any machine that holds a complete copy of one or more *databases*. We use the term "database" loosely to denote a collection of data items; whether such data is managed as a relational database or simply stored in a conventional file system is left unspecified in the architecture. Clients are able to access data residing on any server to which they can communicate, and conversely, any machine holding a copy of a database, including personal laptops, should be willing to service read and write requests from other nearby machines.

We expect that portable computers will be servers for some databases and clients for others. A commonly occurring case may be several users disconnected from the rest of the system while actively collaborating; a canonical example is a group of colleagues taking a business trip together. Rather than giving the members of this disconnected working group access to only the data that they had the foresight to copy to their personal machine, the Bayou design lets any group member have access to any data that is available in the group.

Thus, the Bayou architecture differs from systems like Coda [23][17] that maintain a strong distinction between servers, which hold databases or file volumes, and clients, which hold personal caches. Permitting "lightweight" servers to reside on portable machines is similar to the approach taken to support mobility in Lotus Notes [16] or Ficus [12].

Goal: High availability for Reads and Writes.

Design: Read-any/write-any weakly consistent replication.

Replication is absolutely required in order for non-connected users to access a common database. Many algorithms for managing replicated data, such as those based on maintaining strong data consistency by atomically updating all available copies [4], do not work well in a partitioned network, particularly if site failures cannot be reliably detected. Server-initiated callbacks for cached data invalidation present similar problems. Quorum based schemes [3][10], which can accommodate some types of network partitions, do not work for disconnected individuals or small groups. Algorithms based on pessimistic locking are also unattractive since they severely limit availability [7][8] and perform poorly when message costs are high [6], as is generally the case in mobile environments [1].

To maximize a user's ability to read and write data, even while completely disconnected from the rest of the computing environment, we chose a *read-any/write-any* replication scheme, as was first used in Grapevine [5]. That is, a user is able to read from and write to any copy of the database. We cannot guarantee the timeliness with which writes will propagate to all other replicas since communication with many of these replicas may be currently infeasible. Thus, the replicated databases are only weakly consistent. Techniques for managing weakly consistent replicated data, desired not only for their high availability but also for their scalability and simplicity, have been employed in a variety of systems [5][9][11][16][19].

Goal: Reach eventual consistency while minimizing assumptions about communication characteristics.

Design: Peer-to-peer anti-entropy for propagation of updates.

Servers propagate writes among copies of the database using an "anti-entropy" protocol [9]. This process is often called "reconciliation" when used to synchronize file systems [11][13]. Anti-entropy ensures that all copies of a database are converging towards the same state and will eventually converge to identical states if there are no new updates. To achieve this, servers must not only receive all writes but must also order them consistently.

Peer-to-peer anti-entropy is adopted to ensure that any two machines that are able to communicate will be able to propagate updates between themselves. Even machines that never directly communicate can exchange updates via intermediaries. Each server periodically selects another server with which to perform a pair-wise exchange of writes; the server selected depends on its availability as well as the expected costs and benefits. At the end of this process, both servers have identical copies of the database, that is, they have the same writes effectively performed in the same order. Anti-entropy can be structured as an incremental process so that even servers with very intermittent or asymmetrical connections can eventually bring their databases into a mutually consistent state.

Goal: System support for detection of update conflicts.

Design: Dependency checks on each write.

Because clients may make concurrent writes to different servers or may attempt to update some data based on reading an out-of-date copy, update conflicts are unavoidable in a read-any/write-any replication scheme. These conflicts have two basic forms: *write-write conflicts* in which two clients update the same data item (or sets of data items) in incompatible ways, and *read-write conflicts* in which a client updates some data based on reading the value of another data item that is being concurrently updated by a second client (or was previously updated on a different server than the one being read) [8].

Version vectors, as developed for Locus [21], or simple timestamps are popularly used to detect write-write conflicts [11][13][14][23]. Read-write conflicts can be detected by recording and later checking an application's read-set [8]. These techniques ignore the applications' semantics. Consider a calendar manager in which users interactively schedule meetings by selecting blocks of time. A conflict, as viewed by the application, does not occur simply because two users concurrently edit the file containing the calendar data, but rather conflicts arise if two users schedule meetings at the same time involving the same attendees.

The Bayou system detects update conflicts in an application-specific manner. A write conflict occurs when the state of the database differs in an application-relevant way from that expected by a write operation. A write operation includes not only the data being written or updated but also a *dependency set*. The dependency set is a collection of application-supplied queries and their expected results. A conflict is detected if the queries, when run at a server against its current copy of a database, do not return the expected results.

Bayou's dependency sets can provide traditional optimistic concurrency control by having the dependency queries check the version stamps of any data that was read and on which the given update depends. However, the dependency checking mechanism is more general than this and can, for example, permit "blind" writes where a client does not have access to any copy of the database yet wishes to inject a database update assuming that some condition holds. An example of this is a client that, from his laptop, wishes to schedule a meeting in a particular room, assuming that the room is free at the desired time, but does not currently have a copy of the room's calendar.

Goal: Application-specific resolution of update conflicts.

Design: Merge procedure passed with each write to automatically resolve conflicts.

The system, along with detecting update conflicts, must provide means for resolving such conflicts. One approach often taken in database systems with optimistic concurrency control is to simply abort a conflicting transaction [8]. Other systems rely on humans for resolving conflicts as they are detected. Human resolution is problematic in a mobile computing environment since a user may submit an update to some server and then disconnect while the write is propagating in the background via anti-entropy; at the time a write conflict is detected, i.e. the dependency check fails, the user may be inaccessible.

Bayou allows writes to specify how to automatically resolve conflicts based on the premise that there are a significant number of applications for which the order of concurrently issued write operations is either not a problem or can be suitably dealt with in an application-specific manner at each server maintaining a copy of a database. A Bayou write operation includes an application-specific procedure called a *mergeproc* that is invoked when a write conflict is detected. This program reads the database copy residing at the executing server and resolves the conflict by producing an alternate set of updates that are appropriate for the current database contents.

Mergeprocs resemble mobile agents [28] in that they originate at clients, are passed to servers, and are executed in a protected environment so that they cannot adversely impact the server's operation. However, unlike more general agents, they can only read and write a server's database. A mergeproc's execution must be a deterministic function of the database contents and its static data.

Automatic resolution of concurrent updates to file directories has been proposed for some time and is now being employed in systems like Ficus [22] and Coda [18]. These systems have recently added support for application-specific resolution procedures, similar to mergeprocs, that are registered with servers and are invoked automatically when conflicts arise [18][22]. The appropriate resolution procedure to invoke is chosen based on file properties such as the type of the file being updated. Mergeprocs are more flexible since they may be customized for each write operation based on the semantics of the application and the intended effect of the specific write. For example, in the calendar application, a mergeproc may include a list of alternate meeting times to be tried if the first choice is already taken.

In summary, a Bayou write operation consists of a proposed update, a dependency set, and a mergeproc. The dependency set and mergeproc are both dictated by an application's semantics and may vary for each write operation issued by the application. The verification of the dependency check, the execution of the mergeproc, and the application of the update set is done atomically with respect to other database accesses on the server.

Goal: Commit data to a stable value as soon as possible.

Design: Include a primary whose purpose is to commit data and set the order in which data is committed.

Bayou's weak consistency means that a write operation may produce the desired update at one server but be detected as a conflict at another server thereby producing a completely different update as the result of executing its mergeproc. Also, a write's mergeproc may produce different results at different servers since its execution may depend on the current database state. Varying results can arise if the servers have seen different sets of previous writes or if they process writes in different orders. To achieve eventual consistency, servers must not only receive all writes but must also agree on the order in which they apply these writes to their databases. New writes obtained via anti-entropy may need to be ordered before writes that were previously obtained, and may therefore cause previous writes to be undone and reapplied to the server's database copy. Reapplying a write may, in turn, cause it to update the database in a different way than its previous execution. How can a user ever be sure that the outcome of a write it issued has stabilized?

One way to detect stability of a given write is to gather enough information about each server to determine that no other writes exist or will be accepted in the future that might be ordered prior to the write. Unfortunately, the rate at which writes stabilize in this fashion would depend on the rate at which anti-entropy propagates information among all servers. A server that is disconnected for extended periods of time could essentially delay stability and possibly cause a large number of writes to be rolled back later.

The Bayou design includes the notion of explicitly “committing” a write. Once a write is committed, no other non-committed writes will be ordered before it, and thus its outcome will be stable. A write that has not yet been committed is called “tentative”. A Bayou client can inquire as to whether a given write is committed or tentative. One way to commit a write would be to run some sort of consensus protocol among a majority of servers. However, such protocols do not work well for the types of network partitions that occur among mobile computers.

Instead, each Bayou database has one distinguished server, the “primary”, which is responsible for committing writes. The other, “secondary” servers tentatively accept writes and propagate them toward the primary using anti-entropy. As secondary servers contact the primary, their tentative writes are converted to committed writes, and a stable commit order is chosen for those writes by the primary server. Knowledge of committed writes and their ordering propagates from the primary back to the secondaries, again via anti-entropy. The existence of a primary server enables writes to commit even if other secondary servers remain disconnected. In many cases, the primary may be placed near the locus of update activity for a database; this allows writes to commit as soon as possible.

Goal: Permit disconnected clients and groups to see their own updates.

Design: Clients can read tentative data with an expectation that it will be committed with the same effect if possible.

Clients that issue writes generally wish to see these updates reflected in their subsequent read requests to the database and may even issue writes that depend on reading their previous writes. This should hold even if the client is disconnected from the primary copy and the updates cannot be immediately committed. Moreover, to the extent possible, clients should be unaware that their updates are tentative and should see no change when the updates later commit; that is, the tentative results should equal the committed results whenever possible.

The Bayou system allows clients to read tentative data, if they so desire. Essentially, each server maintains two views of the database: a copy that only reflects committed data, and another “full” copy that also reflects the tentative writes currently known to the server. The full copy is an estimation of what the database will contain when the tentative writes reach the primary.

When two secondary servers exchange tentative writes using anti-entropy, they agree on a “tentative” ordering for those writes. This order is based on timestamps assigned to each write by the server that first accepted it so that any two servers with identical sets of writes will order them identically. Thus, a group of servers that are disconnected from the primary will reach agreement among themselves on how to order writes and resolve internal conflicts. This write ordering is only tentative in that it may differ from the order that the primary chooses to commit the writes. However, in the case where no clients outside the disconnected group perform conflicting updates, the writes can and will eventually be committed by the primary in the tentative order and produce the same effect on the committed database as they had on the tentative one.

Goal: Provide a client with a view of the replicated data that is consistent with its own actions.

Design: Session guarantees.

A serious problem with read-any/write-any replication is that inconsistencies can appear even when only a single user or application is making data modifications. For example, a mobile client could issue a write at one server, and later issue a read at a different server. The client would see inconsistent results unless the two servers had performed anti-entropy with one another sometime between the two operations.

To alleviate such problems, we added session guarantees to the Bayou design. A session is an abstraction for the sequence of read and write operations performed on a database during the execution of an application. One or more of the following four guarantees can be requested on a per-session basis:

- *Read Your Writes* - read operations reflect previous writes.
- *Monotonic Reads* - successive reads reflect a non-decreasing set of writes.
- *Writes Follow Reads* - writes are propagated after reads on which they depend.
- *Monotonic Writes* - writes are propagated after writes that logically precede them.

The intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. Previous work on “causal operations” has tried to provide similar guarantees for weakly consistent replicated data, though without the per-application fine-grain control [19]. Session guarantees do not address the problem of isolation between concurrent applications [20].

Practical implementations of the guarantees have been developed in which no system-wide state is maintained and no additional coordination among servers is needed. The amount of per-session state needed to ensure all of the guarantees is small, consisting of only two version vectors. Also, the cost of checking those version vectors against a server's vectors to determine if the server is sufficiently up-to-date is small, and frequently can be amortized over many session operations. Session guarantees and their implementation are described in more detail in a recently published paper [26].

Goal: Permit applications to choose an appropriate point in the consistency/availability trade-off.

Design: Individually selectable session guarantees, choice of committed or tentative data, age parameter on reads.

Different applications have different consistency requirements and different tolerances for inconsistent data. For this reason, Bayou permits applications to choose just the session guarantees that they require. The main cost of requesting session guarantees is a potential reduction in availability since the set of servers that are sufficiently up-to-date to meet the guarantees may be smaller than all the available servers.

Second, applications may choose between committed and tentative data. Those applications that are unprepared to deal with tentative data and its inherent instability may limit their read requests to only return committed data. This choice is similar to the strict and loose read operations provided in Tait and Duchamp's file system [25].

Finally, applications can specify an age parameter for their reads to ensure that they see committed data in a timely fashion. This parameter might implicitly affect the rate at which secondary servers perform anti-entropy with the primary. It provides clients with a type of bounded inconsistency that resembles quasi-copies [2].

Goal: Give users ultimate control over the placement and use of databases.

Design: Fluid replication in which the number and locations for a database can vary over time as can its primary server.

The Bayou system uses "fluid" replication for managing copies of a database. That is, database copies are allowed to "flow" around in the system changing their degree of replication and their locations. The number of servers (or copies) can vary over time. It can be specified by clients, as well as possibly being determined by the system based on usage patterns and network characteristics. For example, a user with a database on his laptop is free to pass a copy of this database to another user's machine. thereby creating a new server for the database. The primary server for a database may also be changed. Dynamic replication is important in a mobile environment to deal with anticipated network disconnections and to minimize communication costs [1][15].

3. Conclusions and status

The Bayou architecture supports shared databases that can be read and updated by users who may be disconnected from other users, either individually or as a group. Many of the individual design choices are similar to those taken in previous systems for similar reasons. Our contribution is in taking a fresh, bottom-up and critical look at the requirements of mobile computing applications and in carefully pulling together both new and existing techniques into an architecture that meets these requirements. Our emphasis is on supporting application-specific conflict detection and resolution and on providing application-controlled inconsistency. We make minimal assumptions about the sorts of communication capabilities available on mobile computers and about the pattern of network partitions and re-merging that might occur. The motivation for this work arose from our experiences at Xerox PARC with wireless networks and portable devices that were developed to explore our ubiquitous computing vision [27].

The Bayou architecture outlined in the paper has not been fully implemented, though an implementation is currently underway. We are initially building clients and servers that run on SparcStations running Unix and on 486-based subnotebooks running Linux; clients for other types of devices, such as the ParcTab [24] will likely follow. Our database provides a relational model while the query language used in read operations, dependency checks, and mergeprocs is a subset of SQL. The first Bayou application, a meeting room calendar manager and scheduler, has recently been completed, linked with our client stub implementation, and tested against a rudimentary server. We anticipate that experience obtained through building and using applications such as this one will cause the architecture and implementation to evolve into a practical artifact.

4. Acknowledgments

The Bayou project goals and system design have benefited from conversations with a number of colleagues, especially Tom Anderson, Mary Baker, Brian Bershad, Helen Davis, Hector Garcia-Molina, Dan Greene, Carl Hauser, David Nichols, Dan Swinehart, Terri Watson, and Mark Weiser. Atul Adya, a summer intern from MIT, implemented the current Bayou client stub and the meeting room scheduling application.

5. 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] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems* 15(3):359-384, September 1990.

[3] D. Barbara and H. Garcia-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing* 1:119-132, 1986.

[4] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems* 9(4):596-615, December 1984.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] D. K. Gifford. Weighted voting for replicated data. *Proceedings Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979, pages 150-162.

[11] 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. *USENIX Conference Proceedings*, pages 63-71, USENIX, June 1990.

[12] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. *Proceedings Second Workshop on the Management of Replicated Data*, Monterey, California, November 1992, pages 2-5.

[13] J. H. Howard. Using reconciliation to share files between occasionally connected computers. *Proceedings IEEE Workshop on Workstation Operating Systems*, Napa, California, October 1993, pages 56-60.

[14] L. B. Hutson and P. Honeyman. Disconnected operation for AFS. *Proceedings USENIX Mobile & Location-Independent Computing Symposium*, Cambridge, Massachusetts, August 1993, pages 1-10.

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

[16] L. Kalwell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. *Proceedings Conference on Computer-Supported Cooperative Work*, Portland, Oregon, September 1988.

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

[18] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. *Proceedings IEEE Workshop on Workstation Operating Systems*, Napa, California, October 1993, pages 66-70.

[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] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* 28(2):81-87, April 1994.

[21] 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.

[22] 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.

[23] 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.

[24] B. N. Schilit, N. Adams, R. Gold, M. M. Tso, and R. Want. The ParcTab mobile computing system. *Proceedings IEEE Workshop on Workstation Operating Systems*, Napa, California, October 1993, pages 34-39.

[25] C. D. Tait and D. Duchamp. Service interface and replica management algorithm for mobile file system clients. *Proceedings First International Conference on Parallel and Distributed Information Systems*, December 1991, pages 190-197.

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

[27] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM* 36(7):74-84, July 1993.

[28] J. E. White. Mobile agents make a network an open platform for third-party developers. *IEEE Computer*, “Hot Topics” column, November 1994, pages 89-90.