Optimistic Data Replication for Mobile Applications

Michael Coglianese (mlc@cs.brown.edu)

December 19, 2000

1 Introduction

The number of mobile devices in use is exploding, and mobile devices are becoming more and more interconnected. As a result, users are demanding mobile access to their favorite applications as well as to new collaborative and group applications made possible by ad hoc networks of mobile clients. Many of these applications need to access data that is shared between multiple mobile clients.

Optimistic data replication has emerged as the natural way to allow mobile clients with limited network connectivity to access global application data. In optimistic schemes, each machine keeps a cache of the data, and it reads and writes to this data without contacting other machines. Machines then communicate with each other to share their updates, resolving concurrent updates as needed. Optimistic replication has been used to manage data consistency among groups of well-connected servers, but the connectivity of devices in a mobile environment can often vary. In addition, mobile clients have a limited amount of memory, so solutions must ensure that cache sizes are kept to a minimum.

In this paper, I will examine several existing systems that use optimistic data replication to support mobile applications. In my analysis, I will put an emphasis on the design decisions behind the system. These design decisions include the following general areas:

- **System architecture and communication.** One design decision is the degree of communication that is permitted between different machines. The system could employ a *client/server* architecture, where mobile clients can communicate only with an application server, or it could

use a *peer-to-peer* setup where mobile clients are able to intercommunicate. What assumptions are made about the machines in the system's architecture? Are group multicast or broadcast operations employed?

- **Cache management.** Another major issue is deciding how each machine represents and manages its cache of global data. In particular, what additional information needs to be stored with this data in order to implement the system's consistency claims? How are updates to data represented and managed?

- **Update propagation and reconciliation.** A central issue in systems that employ optimistic data replication is how cache writes are sent to the rest of the machines in the system. Inherent in this is how conflicting updates are reconciled. If two updates conflict, must one of them be rejected or can they be somehow merged? Does the system work well only when updates are commutative, meaning that they can be applied in any order?

- **Application involvement.** In some cases, the system allows the mobile application to take part in the system's cache management and update policies. The most common instance of application involvement is in the reconciliation of conflicting updates. In this case, to what degree can the system and application decide how to integrate the updates without resorting to user intervention? What other system policies can the application affect?

For each system I will analyze the advantages, disadvantages, and consequences of those design decisions. This analysis will include an evaluation of how well the systems work with respect to appropriate metrics. These metrics include the effect on mobile application design, the number of conflicts and the expected success of conflict resolution, scalability, and the bandwidth and disk space needed for cache storage and management.

2 Roam

The first system providing optimistic replication that I will discuss is Roam [10, 13]. Similar to its colleagues Ficus [5] and Rumor [2], Roam is peer-to-peer, user-level process that manages shared

file system access. The system uses *selective replication*, meaning that each client's cache contains a subset of the file system. The file system is divided up into volumes, where each volume represents a collection of directories, perhaps a subtree of the file system. Roam does not require that a mobile client cache an entire volume; rather, it allows specific files in a volume to be cached. Roam's only restriction is that if a file is in the cache, so must be its parent directories in the hierarchy.

Peer-to-peer communication is managed in Roam through the *ward* model [12]. A ward consists of a group of mobile clients that are geographically near to each other. All clients in a ward can communicate with each other on a peer-to-peer basis, although such communication may be intermittent or of low quality. A client can potentially be part of multiple wards. Wards form a two-tiered hierarchy; each ward has a *ward master* that connects its ward with a higher level ward. The ward master knows what data is in the replicas of clients in its ward, though it does not actually contain that data. Wards and ward masters are created, managed, and destroyed dynamically throughout the life of the system.

To manage consistency between clients in a ward (or between ward masters at the top level), the clients communicate in an adaptive ring topology. Essentially, clients communicate only with their neighbors on the ring to receive updates to shared data. Communication is one-way: a target replica pulls all updates from a source replica. The target learns about all the updates that the source had made or received, but the source learns nothing of the target. Since the clients are arranged in a ring, updates are passed along transitively between clients which may never have had direct communication. The "adaptivity" of the ring allows it to bypass clients that leave the ward or are temporarily not responding.

The main advantage of using a ring topology to exchange updates is to limit the number of messages sent between clients in a ward. Although all clients can technically communicate with each other, actually using this ability would cause the number of messages to be the square of the number of clients. In a ring, there are only a linear number of messages. As the authors note, however, a single ring topology does not work when combined with selective replication, since neighboring machines on the ring will likely not have the same files in their replicas. Therefore, Roam uses multiple rings between a group of clients - one ring for each file in common in the group.

Rings that have common group members are the combined into a ring that manages all of those files. This means that there could be a large number of rings formed, which would incur a moderate amount of overhead. However, the apparent assumption is that the variation between most clients will either be minimal (clients have most of the same files in common) or complete (a client has none of the same files in common, or has different volumes altogether). In these cases, which are probably the most common usage scenarios, the number of rings would be minimzed.

The reconciliation process between a source and target involves a couple steps. First, for each file in its local cache, the target asks the source whether it has more recent updates to that file. To determine whether a file is more recent or not, Roam maintains *version vectors* [6, 11] for each file in the replica. In this approach, Roam maintains a vector v, where $|v|$ equals the number of replicas in the system. v_i contains the number of updates made or current revision of the file at replica i. If the source has the same or more recent updates from all replicas than the target does, then the source file is said to *dominate* the target. When one file dominates the other, there is no conflict. If neither dominates, then the two replicas conflict on this file, so the file is marked as such, and the system resorts to automatic resolution [14]. Some types of conflicts may be resolved, and Roam (relying on its roots in Ficus) provides different resolvers depending on the file type at issue. User involvement is needed in some cases through the use of system-provided tools.

Simulations of this system reveal that the ward model with two-tier hierarchy, along with selective replication, provides decent scalability. The addition of multiple wards affects only the ward master, since it must connect to those wards in the top level of the hierarchy. However, non-master clients within a ward are not burdened by the communication in other wards. The number of replicas within a ward can not grow too large, however, because it increases communication overhead during the reconciliation procedure. But this a main reason for using the wards in the first place, so this limit is not a major concern.

Clearly, one issue here is the number of reconciliations that an update must go through until it is guaranteed to be received by all other replicas in all wards. The authors analyze that, given M replicas equally distributed between N wards, an update must go through $\frac{M}{N} + \frac{N-3}{2}$ reconciliations. Depending on the frequency of reconcilations between replicas, this is reasonable for a file system.

However, it may not be as reasonable if the same algorithm is implemented in other applications, such as games or collaborative software.

3 Bayou

Bayou [15, 7, 8] is system for managing optimistically replicated data on a group of machines to support collaborative applications. Like Roam, Bayou uses a peer-to-peer model for communication. Unlike Roam, which uses a two-tiered hierarchy and ring-based communication, Bayou imposes no restrictions on that communication - any machine actually may talk to any other one. Bayou leaves the intercommunication pattern of machines up to the application layer. This has the advantage of allowing the application to optimize communication given its specific needs. However, whereas Roam can make some guarantees about the time needed to propagate an update through the system, Bayou can not. Bayou relies on *eventual consistency*, which means that if machines keep exchanging their updates, they will eventually become synchronized.

Each machine maintains a log of all the writes made on that machine or received from others, along with a database that is created by executing those writes in order. A write consists principally of a series of updates to the database. Whereas Roam requires that files be sent across the network, Bayou only needs to transfer these write operations. This is analagous to sending an entire table versus sending only the added or deleted records - obviously an advantage for Bayou. When a write is first made, the write is assigned a timestamp containing the time and server that it was created on.

Writes are initially considered to be *tentative*, meaning that they may conflict with other writes in the system and therefore need to be rolled back. One machine is designated as the primary replica. When a write is added to the primary replica's write log, it is said to be *committed*, and its position in the write log never changes after that point. An important property of Bayou is that, while different machines may have different writes in their logs, all writes are listed in the same order on all machines. Commited writes are listed first, given in the order they appear in the primary replica. Next, tentative writes are listed, given in the order of their timestamps.

Sharing of updates between two machines is similar to that in Roam. Communication is a

pair-wise one-way operation, with a target and source. When the target connects with the source, it receives all writes held by the source that are later than the writes held by the target. It sends the writes in order from the least to most recent. In order to know which are the most recent writes that a machine has, each machine maintains a version vector containing the timestamp of the most recent write received from every other machine. Using version vectors and this communication process, Bayou guarantees that when a machine has an write W_i created on machine S_i, it also has all earlier writes created on machine S_i. Therefore, each write is transmitted to a machine only once.

Bayou supports application-specific conflict detection and management [15]. In addition to its database modifications, each write also contains a dependency check procedure and a merge procedure. When a write is inserted into a machine's write log, it runs the dependency check to see if the addition of the write had the expected effect. The authors give the example of a room reservation system where the write is to reserve a room for one hour. In this case, the dependency check would make sure that the room was actually free during that hour. If the dependency check fails, then the merge procedure is run to see if the write can be performed in some other way. Using an application-defined, arbitrary merge procedure in the case of conflicts enables a great deal of flexibility. In the room reservation example, the merge procedure could try to reserve an different room, or perhaps schedule the same room at a different time.

Since the Bayou architecture is similar to Roam's in a number of ways, it similarly is a fairly scalable system. Since synchronization is a pair-wise operation, the time needed to do this is relative only to the number of writes in the system. Furthermore, version vectors ensure that each write is sent to a machine only once throughout the life of the system. The addition of more replicas affects only the size of the version vectors, and not the amount of time needed to share writes. The use of committed writes allows machines to truncate their write logs, since the position of committed writes in the log never changes.

Another main advantage is Bayou's flexibility. Bayou allows arbitrary merge procedures and dependency checks, and it allows the application layer to define the communication topology between replicas. Unlike Roam, the system does not suffer when communication between machines is lost,

since the only requirement is that machines are able to occasionally communicate. This flexibility in communication presents a drawback as well. By only guaranteeing that writes will eventually be delivered everywhere, machines can not be sure when they will end up seeing a particular write. This burden is placed on the application layer instead.

4 Rover

The Rover toolkit [3] is a system for supporting data sharing in mobile applications. Unlike the two systems described above, Rover uses a client/server architecture. A Rover application uses a set of servers that maintain the data and clients that store a cache of that data. Clients are not permitted to communicate with other clients. Global data in Rover is modeled as a set of *relocatable dynamic objects* (RDOs). RDOs contain both arbitrary application data and application code. RDO application code can be run on both clients and servers.

Like Bayou, Rover uses the concept of tentative and committed updates. The RDO's home server stores the primary copy of the RDO, and clients download RDOs to their local caches. A client can then make tentative updates to the RDO data in its cache regardless of its outside connectivity. Updates are stored as both the modified data and the method invocation. These tentative updates are then sent back to the server whenever connection is established. To transfer RDOs back and forth, Rover uses *queued remote procedure calls* (QRPCs). When a client wants to send an RDO to the server, it adds this request to the QRPC log, sending it when the server becomes available. The same thing happens on the server side: if the server needs to send a reply to a client request, the server will periodically try to contact the client if it is temporarily unavailable.

Servers maintain data consistency when conflicting tentative updates are received from clients. Rover allows the application layer to detect and resolve these conflicts. To assist the application in this process, Rover keeps a wealth of information about each RDO, including version vectors and method call and data modification logs. It is up to the application code running on the server to resolve these conflicts using this information. Once the tentative update is processed by the server, it is committed, and the new RDO will be sent out in response to any future client requests.

One main feature of the Rover system is the vast flexibility given to applications to determine

system policies. First, applications can decide whether to run RDO code on either the client or on the server. This allows resource-poor mobile clients to perform computationally expensive tasks more efficiently. Also, Rover provides support for any consistency model the application wants to implement. While this allows any number of schemes (including pessimistic locking), the authors found that only the primary copy with tentative update model was useful for their applications. One way in which the system is not flexible is that it does not support client-to-client communication. This limits the scope of the applications that can be built using Rover; indeed, the applications they developed (stock market watcher, email reader) are not especially collaborative in nature.

The use of RDOs and QRPCs improves performance in a couple ways. Multiple tentative updates that need to be sent to the server can be batched in one QRPC instead of being sent immediately. This reduces communication costs for both the client and the server. Also, QRPCs are asynchronous, so the application on the mobile client doesn't need to block while waiting for the server's response. Since connectivity is often slow in a mobile environment, this further improves performance. The ability to ship RDOs to servers to perform expensive tasks also improves system performance in terms of CPU usage, but it uses up network bandwidth.

5 Conclusions

All three of these systems support optimistic data replication in a variety of mobile scenarios, and all three do it somewhat differently. But one feature that these systems have in common in some kind of application involvement and customization. Not only do most mobile applications need to be aware of the resource and networking limitations of mobile devices, but they also need to be able to affect the consistency policies implemented in the system, Flexible conflict detection and resolution is vital for proper execution of future mobile applications.

Another main contrast between these systems was the types of permitted communication between machines. Currently, simple client/server synchronization of data is implemented in a number of popular applications, such as Microsoft's ActiveSync for Windows CE [4]. In the future, as wireless connectivity between mobile devices will become increasingly viable, mobile devices will be able to form ad hoc networks. In these networks, peer-to-peer communication will be vital. Therefore

	Roam	Bayou	Rover
Target applications	file system	collaborative apps	mobile apps with disconnection operation
Architecture	peer-to-peer, two-tiered hierarchy, adaptive ring in group	peer-to-peer, arbitrary	client/server
Data representation	files	write log	objects
Reconciliation	one-way	one-way	two-way with deferred communication
Conflict detection	file modification	RDO modification	application-defined procedure
Conflict resolution	application-defined, static	application-defined, dynamic	application-defined, dynamic

Figure 1: Comparison of key features of the three systems.

I would expect future research in distributed systems for mobile applications to move away from using strict client/server architectures. Whereas Bayou originally envisioned that updates could be transferred on floppy disks between machines, future wireless technology will make its peer-to-peer model even more suited to the task.

References

[1] M. Coglianese. Mobile Aleph: A System for Distributed Mobile Applications. Master's thesis, December 2000.

[2] R. Guy, P. Reicher, D. Ratner, M. Gunter, W. Ma, and G. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-peer Replication. In *Proceedings: ER'98 Workshop on Mobile Data Access*, 1998.

[3] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile Computing with the Rover Toolkit. In *IEEE Transactions on Computers: Special issue on Mobile Computing*, 46(3), March 1997.

[4] T. Ogasawara. ActiveSync 3.1 Tricks, Tips, & Tweaks. http://www.microsoft.com/mobile/pocketpc/stepbystep/activesync.asp.

[5] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on Optimistically Replicated, Peer-to-Peer Filing. Software – Practice and Experience, December 1997.

[6] D. S. Parker, Jr., G. 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. In *IEEE Transactions on Software Engineering*, May 1983.

[7] K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Bayou: Replicated Database Services for World-wide Applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, September 1996.

[8] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[9] Y. Saito. Optimistic Replication Algorithms. Technical report, August 2000.

[10] D. Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing. PhD thesis, UCLA, January 1998.

[11] D. Ratner, P. Reiher, and G. Popek, Dynamic Version Vector Maintenance. UCLA Technical Report CSD-970022, June 1997.

[12] D. Ratner, G. Popek, and P. Reiher. The Ward Model: A Scalable Replication Architecture for Mobility. In *Workshop on Object Replication and Mobile Computing*, October 1996.

[13] D. Ratner, P. Reiher, G.J. Popek and R. Guy. Peer Replication with Selective Control. In *MDA '99: First International Conference on Mobile Data Access*, December 1999.

[14] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Conference Proceedings*, June 1994.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.