Flexible Update Propagation for Weakly Consistent Replication

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

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

Abstract

Bayou's anti-entropy protocol for update propagation between weakly consistent storage replicas is based on pair-wise communication, the propagation of write operations, and a set of ordering and closure constraints on the propagation of the writes. The simplicity of the design makes the protocol very flexible, thereby providing support for diverse networking environments and usage scenarios. It accommodates a variety of policies for when and where to propagate updates. It operates over diverse network topologies, including low-bandwidth links. It is incremental. It enables replica convergence, and updates can be propagated using floppy disks and similar transportable media. Moreover, the protocol handles replica creation and retirement in a light-weight manner. Each of these features is enabled by only one or two of the protocol's design choices, and can be independently incorporated in other systems. This paper presents the anti-entropy protocol in detail, describing the design decisions and resulting features.

1. Introduction

Weakly consistent replicated storage systems with an "update anywhere" model for data modifications require a protocol for replicas to reconcile their state, that is, a protocol to propagate the updates introduced at one replica to all other replicas. A key advantage of weakly consistent replication is that, by relaxing data consistency, the protocol for data propagation can accommodate policy choices for *when* to reconcile, *with whom* to reconcile, and even *what* data to reconcile. In this paper we present Bayou's **anti-entropy** protocol for replica reconciliation. The protocol, while simple in design, has several features intended to support diverse network environments and usage scenarios. The contribution of this paper is to demonstrate how the anti-entropy design, based on pairwise-communication between replicas and the ordered exchange of update operations stored in per replica logs, enables this set of features and functionalities:

- *Support for arbitrary communication topologies*: the protocol provides the mechanism to propagate updates between any two replicas. In turn, the theory of epidemics ensures that these updates transitively propagate throughout the system [3].
- *Operation over low-bandwidth networks*: reconciliation is based on the exchange of update operations instead of full database contents, and only updates unknown to the receiving replica are propagated.
- *Incremental progress*: the protocol allows incremental progress even if interrupted, for example, due to an involuntary network disconnection.
- *Eventual consistency*: each update eventually reaches every replica, and replicas holding the same updates have the same database contents.
- *Efficient storage management*: the protocol allows replicas to discard logged updates to reclaim storage resources used for reconciliation.
- *Propagation through transportable media:* one replica can send updates to another by storing the updates on transportable media, like diskettes, without ever having to establish a physical network connection.
- *Light-weight management of dynamic replica sets*: the protocol supports the creation and retirement of a replica through communication with only one available replica.
- *Arbitrary policy choices*: any policy choices for when to reconcile and with which replicas to reconcile are supported by the anti-entropy mechanism. The policy need only ensure that there be an eventual communication path between any pair of replicas.

Other weakly consistent replicated systems support subsets of these functionalities. For example, Coda's reconciliation protocols allow server replicas to reconcile

---

* Now at Oracle Corporation, 500 Oracle Way, Redwood Shores,

with each other, and mobile replicas to reconcile with servers, but mobiles cannot reconcile amongst themselves [11]. In Ficus, reconciliation can occur between any pair of replicas, however server creation and retirement requires coordination among all replicas [7]. Oracle 7 has a two-level hierarchy of replicas: master replicas send their transactions to all other masters, but cannot forward transactions received from other masters; a snapshot replica can only reconcile with its specific master, independently of the availability of other masters [16]. Gray *et al.* also proposed a two-tier replication model that, in contrast to Oracle's system, ensures convergence of the replicas but does not allow reconciliation between mobile replicas [6]. Golding's time-stamped anti-entropy protocol [4] comes closest to Bayou's. Many of the mechanisms in his design are similar, however he suggests a heavier weight mechanism to create replicas and a less aggressive approach for replicas to reclaim storage resources.

The Bayou system places additional requirements on its anti-entropy protocol due to its support for conflict detection and resolution based on per-write dependency-checks and merge procedures [20] and for session guarantees [19]. By presenting the protocol in detail, along with the design decisions that went into it, this paper shows how the protocol design supports both these requirements of the Bayou system, as well as the features listed above.

We believe that all of the features enabled by Bayou's anti-entropy protocol are important. First, because applications and users have different requirements for data reconciliation, the protocol supports the replica's ability to choose when to reconcile and with whom to reconcile. For example, users of personal information management applications, like address books and calendars, can reconcile their databases differently than enterprise-wide databases, like the ones used for intranet web-sites. Communication can therefore occur at "convenient" times. Second, the protocol was designed to effectively support the variety of networking and computing environments these applications and users may operate in.

The paper starts with a simple protocol for anti-entropy, highlighting the features enabled by this basic design: support for arbitrary networking environments, support for low bandwidth networks, and incremental progress. It then describes protocol extensions that enable other desired features: management of the storage resources required for the operations log, propagation using transportable media, support for eventual consistency, and light-weight management of dynamic replica sets. The paper concludes with a general discussion of how the anti-entropy protocol's features can be implemented in other systems, potential drawbacks of the protocol, policy choices enabled by the protocol, performance measurements, and an expanded discussion of related work.

2. Basic Anti-entropy

The goal of anti-entropy is for two replicas to bring each other up-to-date. In Bayou, the storage system at each replica, also called a server, consists of a ordered log of updates, called *writes*, and a *database* that results from the in-order execution of these writes. A server's *write-log* contains all writes that have been received by that Bayou server either from an application or from other servers. Therefore, anti-entropy needs to enable two servers to agree on the set of writes stored in their logs.

For the purpose of this paper, a Bayou write can be thought of as a procedure that generates a set of updates to be applied at the database. Specifically, a Bayou write consists of three components: a set of updates, a dependency check, and a merge procedure. The dependency check and the merge procedure of a write let each server that receives the write decide if there is a conflict and, if so, how to resolve it [20].

When a Bayou server first receives a write from a client application, the server assigns a monotonically increasing *accept-stamp* to the write. Accept-stamps can be time-stamps or simple generation counters. As it propagates via anti-entropy, each write carries its accept-stamp and the identifier of the server that assigned the stamp. Accept-stamps define a total order over all writes accepted by a server and a partial order, which we call the *accept-order*, over all writes in the system. Write A precedes write B in the accept-order when both were accepted by the same server and write A was accepted before write B. Servers store writes in their write-logs in an order that is consistent with this accept-order.

The simplest anti-entropy protocol can now be described. The protocol is based on the following three design choices for the reconciliation process:

1. it is a one-way operation **between pairs of servers**;
2. it occurs through the **propagation of write operations**;
3. write propagation is **constrained by the accept-order**.

Pair-wise communication supports the reconciliation of any two servers independently of which other servers may be available and of how the network connection between the servers is established. The protocol relies on the theory of epidemics to ensure that writes eventually propagate to all other replicas [3].

A Bayou server can choose its anti-entropy partner at random or based on other knowledge, like network characteristics. In fact, ad-hoc network connections between arbitrary replicas, as possible with wireless infrared links, can be easily supported. Alternatively, a system could choose to force more structure on the communication patterns between replicas, for example, by designating master replicas and subordinate replicas that only reconcile with their masters or by organizing replicas into logical reconciliation rings. Structured communication

```
anti-entropy(S,R) {
    Get R.V from receiving server R
    # now send all the writes unknown to R
    w = first write in S.write-log
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            # w is new for R
            SendWrite(R, w)
        w = next write in S.write-log
    END
}
```

Figure 1. Basic anti-entropy executed at server S to update receiving server R

patterns permit accurate information about the state of the replicas to be maintained more easily and to be used to optimize communication between the replicas. However, by restricting the set of servers with which to communicate, update propagation is more likely to suffer from communication outages. We opted for the peer-to-peer reconciliation model because of the variety of possibly changing communication topologies it supports.

The pair-wise anti-entropy protocol was designed to be uni-directional. One server brings another one up-to-date by propagating those writes not yet known to the receiving server. The advantage of one-way reconciliation is that the process only requires an initial exchange of state information, thereafter all the protocol's state is kept at the sending replica and communication flows in only one direction, from the sender to the receiver.

The anti-entropy design is based on the exchange of write operations because Bayou's conflict detection and resolution mechanisms require that writes are executed at all replicas. Propagating operations, instead of database contents, has other advantages. Namely, the amount of data propagated during reconciliation is proportional to the update activity at the replicas instead of being dependent on the overall size of the data being replicated. Thus, when the database size is much larger than the database updates, the bandwidth required for the execution of the protocol is reduced. Furthermore, the propagation of update operations avoids any ambiguity introduced by the creation and deletion of replicated objects. Protocols based on the exchange of deltas or differences in data values require additional mechanisms to correctly handle this ambiguity because the existence of a value at one replica and the lack thereof at another cannot correctly identify whether the value is new or it has been deleted. Finally, write operations can easily be stored in a log, which can then be used during reconciliation to decide which operations need to be propagated. Aside from the creation/deletion ambiguity, protocols based on deltas have properties similar to those of protocols based on the propagation of update operations.

Our third design choice, enforcing the partial accept-order during anti-entropy, is necessary to maintain a closure constraint on the set of writes known to a server, which we call the *prefix-property*. The prefix property states that a server R that holds a write stamped Wi that was initially accepted by another server X will also hold all writes accepted by X prior to Wi. The prefix-property enables the use of version vectors to compactly represent the set of writes known to a server. More precisely, the entry for another server X in R's version-vector, R.V(X), is the largest accept-stamp of *any* write known to R that was originally accepted from a client by X.

The basic anti-entropy algorithm, shown in Figure 1, updates the receiving server R with the writes stored at sending server S. This initial protocol assumes that servers retain all writes they have ever received. This simplifying, but impractical, assumption is later relaxed in section 3. During anti-entropy, the prefix property and the ensuing use of version vectors enable a server to correctly determine which writes are unknown to the receiving server R by comparing the accept-stamp of a write in its write-log with the entry corresponding to the write's accepting server in R's version-vector. The algorithm demonstrates the incremental transmission of each unknown write from S to R. The reverse process, to update S from R, is identical.

The algorithm is very simple. The sending server gets the version vector from the receiving server; then it traverses its write-log and sends the receiving server each write not covered by that vector. It is worth pointing out that the protocol traverses the sender's write-log only once.

A feature of this algorithm is that it allows anti-entropy to be incremental. In other words, reconciliation between two replicas can make progress independently of where the protocol may get interrupted due to network failures or voluntary disconnections. When a new write arrives at the receiver it can be immediately included in the receiver's write-log because the sending replica ensures that the receiving server will hold all writes necessary to satisfy the prefix property. If interrupted while sending writes, those writes transmitted successfully to the receiving server can thus be processed and stored in the receiver's write-log. Most importantly, during the next execution of the protocol, these writes need not be resent and the sending server only propagates those writes still unknown to the receiving server. Since the ordering in which the writes reach the receiving server is important to ensure the prefix property, the anti-entropy protocol needs to be implemented over a transport layer that guarantees ordered delivery of messages.

The basic anti-entropy algorithm has several of the features we deem important in a reconciliation protocol: it supports a variety communication topologies, it supports a variety of policy choices for when and with whom to reconcile, it operates over low bandwidth networks, and it makes incremental progress in the presence of protocol interruptions. Additionally, as shown in section 4, the protocol's incrementality and pair-wise nature make it adaptable for reconciliation through transportable media, like floppy disks or PCMCIA storage cards, and an extension of the prefix property enables the light-weight management of dynamic replica sets. Before discussing these additional functionalities we focus on relaxing the algorithm's reliance of ever-growing write-logs.

3. Effective Write-log Management

Although very simple, the anti-entropy algorithm presented in Figure 1 is based on a generally unreasonable assumption: that servers do not discard writes from their write-logs. In practice, although disks are continuously becoming cheaper and denser, it is unreasonable to assume that replicas can store ever-growing logs of operations. In particular, mobile hosts do not have unbounded storage. This section shows how servers can effectively manage the storage resources of their write-logs.

Previous work on propagating logged writes observed that a write can be discarded from a replica's log once that write has fully propagated to all other replicas. Determining which writes have fully propagated can be done by running a distributed snapshot algorithm to establish a "cutoff" timestamp [17] or by having replicas maintain an acknowledgment vector [4] or timetable [1, 12, 21] of which replicas have received what writes. The problem with these approaches is that a single, long-disconnected replica can cause the write-logs at all other replicas to grow indefinitely. Sarin and Lynch noted this problem and proposed forcibly removing such sites from the replica set [17].

Bayou takes a different approach. In Bayou, each replica can independently decide when and how aggressively to prune a prefix of its write-log subject to the constraint that only "stable" writes get discarded. The notion of write stability is discussed below. An important consequence of permitting servers to discard writes that may not have fully propagated is that anti-entropy between servers that are too far "out of synch" may require transferring the full database state from one server to the other. Thus, there is a storage-bandwidth tradeoff based on how aggressively replicas prune their logs and how frequently replicas perform anti-entropy. This section, after presenting Bayou's actual anti-entropy protocol with support for write-log truncation, presents a discussion of this tradeoff.

3.1. Write Stability

A *stable write*, also called a *committed write*, is one whose position in the write-log will not change and hence never needs to be re-executed at that server. Any mechanism that stabilizes the position of a write in the log can be used. Details on the benefits and drawbacks of several write stabilizing mechanisms have been described in a previous publication [20].

Bayou uses a primary-commit protocol to stabilize writes, hereby ensuring that the stabilization process does not slow down due to lengthy disconnections of some replicas. In this protocol, one database replica is designated as the *primary* replica and its role is to stabilize (commit) the position of a write in the log when it first receives the write. As the primary commits a write, it assigns a monotonically increasing commit sequence number (CSN) to the write. The CSN is the most significant factor used to determine a write's position in the log; uncommitted or *tentative* writes have a commit sequence number of infinity. The commit sequence numbers and accept-stamps define a new partial order over the writes in the system, where write A precedes write B if A has a smaller CSN, or if both are uncommitted and were accepted by the same server and write A was accepted before write B. In this order committed writes are always totally ordered amongst themselves, are ordered before any tentative writes, and are thereby stable. The CSN information propagates back to all other servers through an extension of the anti-entropy algorithm described below. When a non-primary replica learns of a write's final CSN, the write becomes stable at that server since the replica will previously have learned of all writes with lower commit sequence numbers.

This more complex partial order, called *stable-order*, preserves the prefix property requirement of anti-entropy because: (1) servers reconcile uncommitted writes with the primary using the same protocol described thus far, hence ensuring that the prefix property holds at the time writes are committed, and (2) servers always propagate committed writes before tentative writes as described below. The next subsections show how the anti-entropy protocol changes to support write commitment, and how the stable-order is used to aggressively truncate writes from servers' logs.

```
anti-entropy(S,R) {
    Get R.V and R.CSN from receiving server R
    # first send all the committed writes that R does not know about
    IF R.CSN < S.CSN THEN
        w = first committed write that R does not know about
        WHILE (w) DO
            IF w.accept-stamp <= R.V(w.server-id) THEN
                # R has the write, but does not know it is committed
                SendCommitNotification(R, w.accept-stamp, w.server-id, w.CSN)
            ELSE
                SendWrite(R, w)
            END
            w = next committed write in S.write-log
        END
    END
    w = first tentative write
    # now send all the tentative writes
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            SendWrite(R, w)
        w = next write in S.write-log
    END
}
```

Figure 2. Anti-entropy with support for committed writes (run at server S to update R)

3.2. Propagation of Committed Writes

The part of a server's write-log corresponding to committed or stable writes can be represented by either another version vector, a commit vector, or by the highest commit sequence number known to a server, S.CSN. Since committed writes are totally ordered by their commit sequence numbers and they propagate in this order, the commit sequence number represents the committed portion of the write-log in a concise way. The algorithms in this section will therefore use S.CSN for this purpose.

To propagate the commit information of writes, the anti-entropy algorithm cannot just test whether a write is covered by the receiving server's version vector. The receiving server may have the write, but not know that it is committed. The sending server must therefore first inspect all the committed writes that the receiving server may be missing. As shown in Figure 2, the algorithm starts by comparing the two servers' highest commit sequence numbers. If the sender holds committed writes that the receiver is unaware of, it will send them to the receiver. Notice that for writes that the receiver already has in tentative form but for which it does not know the commit sequence number, only a commit notification is sent. A commit notification only includes the write's accept-stamp, server-id, and new commit sequence number instead of the entire write. After the committed portion of the write log is processed, the same algorithm as before is used to send all the new tentative writes to the receiving server.

3.3. Write-log Truncation

The anti-entropy protocol allows replicas to truncate any prefix of the stable part of the write-log whenever they desire or need to do so. The implication of truncating the write-log is that on occasion a replica's write-log may not hold enough writes to allow incremental reconciliation with another replica. That is, the sending server may have truncated writes from its write-log that are yet unknown to the receiver. This can occur, for example, when the sending server has received and later truncated committed writes that have not reached the receiving replica because the receiving replica has been disconnected for a long time. The protocol needs to detect and handle this possibility.

To test whether a server is missing writes needed for anti-entropy, each server maintains another version vector, S.O, that characterizes the omitted prefix of the server's write-log; a commit sequence number is also maintained for the omitted part of the log. A server can easily detect whether it is missing writes needed to execute anti-entropy with another server if its omitted sequence number, S.OSN, is larger than the other server's commit sequence number, R.CSN. If so, there exist committed writes that the sending server truncated from its log, and that the receiver has not yet received. Under this circumstance, if the two servers still wish to reconcile, a full database transfer has to occur. That is, the receiving replica must receive a copy of the sender's database that includes all writes characterized by the omitted vector. By sending this database the sender

```
anti-entropy(S,R) {
    Request R.V and R.CSN from receiving server R
    #check if R's write-log does not include all the necessary writes to only send writes or
    # commit notifications
    IF (S.OSN > R.CSN) THEN
        # Execute a full database transfer
        Roll back S's database to the state corresponding to S.O
        SendDatabase(R, S.DB)
        SendVector(R, S.O) # this will be R's new R.O vector
        SendCSN(R, S.OSN) # R's new R.OSN will now be S.OSN
    END
    # now same algorithm as in Figure 2, send anything that R does not yet know about
    IF R.CSN < S.CSN THEN
        w = first committed write that R does not yet know about
        WHILE (w) DO
            IF w.accept-stamp <= R.V(w.server-id) THEN
                SendCommitNotification(R, w.accept-stamp, w.server-id, w.CSN)
            ELSE
                SendWrite(R, w)
            END
            w = next committed write in S.write-log
        END
    END
    w = first tentative write in S.write-log
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            SendWrite(R, w)
        w = next write in S.write-log
    END
}
```

Figure 3. Anti-entropy with support for write-log truncation (run at server S to update server R)

makes sure that the receiver knows of all the writes needed to proceed with the regular, more incremental part of the algorithm.

Figure 3 presents the anti-entropy algorithm with support for write-log truncation. The protocol starts by checking if the sender has truncated any needed writes from its write-log. If it has all the entries necessary to only send writes or commit notifications, the algorithm continues just as described earlier. However, if there are missing writes, it sends the contents of the full database to the receiving server in addition to the version vector and the commit-stamp that characterize the database being sent. Once the receiving server receives the database and the corresponding new omitted vector and sequence number, it removes all writes from its write-log that are covered by the new omitted vector, but more importantly, keeps all the writes not covered by this vector, since these may be unknown to the sender. After the database transfer, the algorithm transitions back to incrementally sending the remaining commit notifications and writes not yet known to the receiving replica.

A couple of characteristics of this algorithm should be pointed out. First, sending the complete database during reconciliation may require much more network bandwidth than the incremental, per write, part of the algorithm. Second, the database transfer is not incremental; the receiving server must obtain the full database and the corresponding version vector and commit sequence number for reconciliation to succeed.

3.4. Storage and Networking Resource Tradeoff

Truncating a server's write-log trades off potentially increased usage of network resources with increased storage requirements by one server to bring another server up-to-date. A server either retains sufficient writes to update other servers incrementally, or truncates writes aggressively, which may cause occasional full database transfers. Avoiding a full database transfer is important if servers are synchronizing through low-bandwidth or costly networks and the database is large. Thus, the challenge is to reduce the server's storage resources occupied by the write-log while keeping the chance of having to perform a full database transfer low.

The choice of when to truncate the write-log is left to each server's discretion. One potentially interesting policy would be for the server to maintain running estimates of the rate at which writes are committed and of the rate at which writes propagate through the system, and to use these estimates to establish when and how much of the write-log to truncate. Another, much simpler, policy is to truncate the write-log when free disk-space at the server falls below a certain threshold. Another, more conservative, but potentially more accurate, approach would be to maintain an estimate of the maximum commit sequence number known to all servers.

3.5. Rolling Back the Write-log

The write-log of a server needs to be rolled back, and the effect of the writes undone from the database, in two different situations during anti-entropy: a sender needs to rollback its write-log if a full-database transfer is required, while a receiver has to roll its log back to the position of the earliest write it receives. Rollbacks at the sender's side should be rare, since we expect full database transfers to be rare.

On the receiver's side, the write-log is rolled back at most once per anti-entropy session. Two optimizations can further reduce the overhead of rollback operations. First, if the replica is receiving writes from more than one replica at a time, that is, the server is involved in multiple anti-entropy sessions, the write-log only needs to be rolled back once to the insertion point of the earliest write being received. Second, the receiving server does not need to redo the rolled-back writes until the next read from an application. Hence, there is a tradeoff between lowering the cost of near consecutive anti-entropy sessions and the latency of the next read from a client. A replica could therefore roll its write-log forward, that is, redo the rolled-back writes, when a certain time threshold has passed since an anti-entropy session. Such a threshold can be based on the frequency of read operations.

4. Anti-entropy Protocol Extensions

So far, the paper has presented a reconciliation protocol that supports different networking environments and reconciliation policies, is incremental, and allows servers to manage the storage resources and performance of their write-logs to their best convenience. As mentioned earlier, the simple anti-entropy design also enables additional protocol extensions: server reconciliation using transportable media, support for session guarantees and eventual consistency, and light-weight mechanisms to manage server version vectors when replicas can be created or retired at any time. These features are enabled by the three basic anti-entropy design choices, pair-wise communication, exchange of writes and write propagation according to specific write orders. As described in this section, they also work well with the changes made to the algorithm for more effective storage management.

4.1. Anti-entropy through Transportable Media

In addition to supporting varying networking environments, the anti-entropy protocol easily extends to using transportable media, like floppy disks, PCMCIA storage cards or even PDAs, instead of an actual network connection.

Figure 4 presents an off-line anti-entropy algorithm that outputs information about a server's write-log and database to a file. The main difference between this algorithm and the one discussed in section 3 is that instead of sending the data over a network connection, the updates are stored into a file. This file is later used by another server to incorporate the new updates into its write-log.

The off-line algorithm has additional features not present in the on-line version. First, it takes two parameters, a commit sequence number and a version vector; these parameters define the minimum state required by a potential receiver of the file. Any server whose commit sequence number is at least as large as the CSN parameter and whose version vector dominates the version vector parameter can use the file to update its write-log. Second, the algorithm writes out the commit sequence number and version vector parameters to allow any server that is presented with the file to determine whether it meets the minimal state requirements to use the file. Finally, the algorithm also writes out the sender's commit sequence number, S.CSN, and the full version vector, S.V. By doing so, it enables receiving servers to quickly determine whether the file holds anything new.

The algorithm can be modified to test the space remaining on the auxiliary storage device each time new data is written. When the device fills up the current file is terminated by writing the CSN and V version vector corresponding to the last write included in the file. Then, a new file-anti-entropy session can be started with the closing parameters of the previous session. Thereby sets of devices with files that incrementally update other servers can be generated, a feature that is useful if resource-limited auxiliary devices like floppy disks are used for off-line reconciliation.

4.2. Session Guarantees and Eventual Consistency

In addition to the partial propagation order required by the prefix property, Bayou has two additional ordering requirements: (1) a causal order to provide session guarantees to applications and (2) a total order to ensure eventual consistency of all replicas. This subsection shows

```
file-anti-entropy(fileID, CSN, V) {
    OutputCSN(fileID, CSN);
    OutputVector(fileID,V);
    IF (S.OSN > CSN) THEN
        # Execute a full database transfer
        Roll back S's database to the state corresponding to S.O
        OutputDatabase(fileID, S.DB)
        OutputVector(fileID, S.O) # this will be the receiver's new R.O vector
        OutputCSN(fileID, S.OSN) # the receiver's new R.OSN will now be S.OSN
        CSN = S.OSN; # CSN now points to S.OSN, which will be the receiver's new CSN at this point
    END
    # write anything that is not covered by CSN and V
    IF CSN < S.CSN THEN
        w = first write following the write with commit sequence number = CSN
        WHILE (w) DO
            IF w.accept-stamp <=V(w.server-id) THEN
                OutputCommitNotification(fileID, w.accept-stamp, w.server-id, w.CSN)
            ELSE
                OutputWrite(fileID, w)
            END
            w = next committed write in S.write-log
        END
    END
    w = first tentative write in S.write-log
    WHILE (w) DO
        IF V(w.server-id) < w.accept-stamp THEN
            OutputWrite(fileID, w)
        w = next write in S.write-log
    END
    OutputCSN(fileID,S.CSN);
    OutputVector(fileID,S.V);
}
```

Figure 4. Off-line anti-entropy through transportable media (from S to a file)

how both of these stronger write orders are easily supported by the anti-entropy protocol; in fact, no changes need to be made to any of the algorithms in Figures 1-4.

Bayou provides applications with session guarantees to reduce client-observed inconsistencies when accessing different servers. The description of session guarantees has been presented elsewhere [19]. However, with respect to anti-entropy, the important aspect of session guarantees is that their implementation requires writes to be causally ordered. The causal order is a refinement of the accept-order, called *causal-accept-order*, and specifies that any write A precedes another write B if and only if, at the time write B was accepted by some server from a client, write A was already known to that server. The causal-accept-order is established through the accept-stamps assigned to writes when they are first accepted by a server. To this end, each server maintains a logical clock [13]. This logical clock advances both when new writes are accepted by the server from clients, or when writes with higher accept-stamps are received through anti-entropy. Thus, a write accepted by the server will always get a higher accept-stamp than all other writes known to that server, and through that get ordered after them. Because the ordering constraints of the casual-accept-order are stronger and cover those defined by the regular accept-order, the prefix property continues to hold; furthermore, propagating writes in the causal-accept-order is sufficient for the order to be applied at all servers. The anti-entropy protocol thus supports the causal ordering needed to implement Bayou's session guarantees without changes.

In general, without making assumptions about the commutativity of writes, a total write order is necessary to ensure that replicas holding the same set of writes also hold the same database contents. The stable-order introduced in section 3.1 provides eventual consistency of stable writes. However, we deem eventual consistency to be an important property of weakly consistent storage systems, which should be achieved even for non-stable writes. The accept-order can be easily converted into a total order by using the identification of the server that accepted the write: accept-stamps are used for the causal-accept-order described above and server identifiers are used to break ordering ties between writes with equal accept-stamps. To ensure eventual consistency, writes are propagated between servers and stored in a server's write-log according to the total order defined by the accept-stamp and server-id tuple. The stable-order of section 3.1 can also be converted into a total order. To convert the stable-order into a total order three factors are used, namely <CSN, accept-stamp, server-id>, with the commit sequence number being the most significant factor, and the sever-id again only used to break ties among accept-stamps of non-stable writes.

The ordering imposed on the propagation and execution of writes plays two different roles: (1) ensure the prefix property that enables the version vector representation of a replica's state, and (2) provide applications with guarantees on the "quality" of the data held by a replica. The ordering extensions discussed in this subsection, causal and total, address only the second role. Furthermore, by being consistent with the ordering requirements for the prefix property, no changes were needed to accommodate these extensions in the design of the anti-entropy algorithm itself. In fact, the anti-entropy algorithms can accommodate any order that provides the closure properties required by the prefix property.

4.3. Light-weight Server Creation and Retirement

In most systems, the creation of a data replica is a heavy weight operation. It normally requires the intervention of system administrators or the interaction with specific servers. In Oracle, for example, master replicas can only be created at the master definition replica, and all existing masters must quiesce during the new master's creation; snapshot replicas can only be created at master replicas [16]. Coda requires that the replication factor and the locations of servers be specified at volume creation time; although the system supports later addition of replicas, the implementation is based on the assumption that it will not be a frequent operation [18]. Mobile caches in Coda can only be loaded from replica servers and not from other mobile caches. Golding's mechanisms require that K servers be available for server creation [4], so that K-1 of these servers may fail and at least one server will include the newly created server in its view of what replicas exist.

Lighter weight mechanisms for server creation and retirement enable more flexible usage scenarios. For example, when two colleagues meet on a business trip, one can get a replica of the budget plan from the other colleague and immediately start receiving all the updates made throughout the budgeting process. The user does not have to wait for a connection with either a master server or a quorum of replica servers. Lotus Notes has identified the importance of this feature and advertises it as one of the key differentiators of the system [10].

In Bayou new servers can be created, and similarly retired, by communicating with any available server. Anti-entropy can easily support these operations if the version vectors are updated to include or exclude the new or retired servers. Dynamic management of the version vectors needs a mechanism to (1) uniquely assign identifiers to newly created servers, and (2) allow any server to correctly determine whether a server has been newly created or retired. The prefix property and the causal-accept-order requirements placed on the propagation of Bayou writes are used to provide these mechanisms. Write accept-stamps are used to assign server identifiers that exactly determine the location and time of each server's creation. These server identifiers can then be compared with the version vectors stored at each replica, to determine whether a server is new or has been retired. The next two subsections describe the creation and retirement of servers in more detail.

Creation and Retirement Writes

A Bayou server S_i creates itself by sending a *creation write* to another server S_k. Any server for the database can be used. The creation write is handled by S_k just as a write from a client. The write is inserted in S_k's write log, and is identified with the <infinity, $T_{k,i}$, S_k> three-tuple, where $T_{k,i}$ is the accept-stamp assigned by S_k.

The creation write serves two main purposes. First, as it propagates via anti-entropy, it informs other servers of the existence of S_i. The effect of the execution of the write is that an entry for S_i is added to the server's version vectors that cover this write. Second, it provides S_i with a server-id that is globally unique and clearly identifies the time of its creation. Specifically, <$T_{k,i}$, S_k> becomes S_i's server-id.

The new server also uses the value of ($T_{k,i}$ + 1) to initialize its own accept-stamp counter. This initialization is necessary so all writes accepted by the new server follow its creation write in the causal-accept-order.

Note that the recursive nature of the server identifiers affects the size of the version vectors. At one end, if all servers are created from the first replica for the database, all server identifiers will contain only one level of recursion and thus be short. On the other hand, if replicas are created linearly, one from the next, server identifiers will be increasingly longer, and the version vectors for such a database will therefore also be much larger.

When a server is going to cease being a server for a database, it does so by issuing a *retirement write* to itself. Again, the write is stamped just like any other Bayou write. Its meaning is that the server is going out of service. At this point, the server will no longer accept new writes from clients. However, the server must remain alive until it performs anti-entropy with at least one other server so that all its writes, including its retirement write, get propagated to other servers.

When a server receives a retirement write, it removes the corresponding entry from all the server's version vectors that cover this write. The prefix property ensures that a server S_k will have received and processed all writes accepted by S_i before removing S_i from its version vector.

Logically Complete Version Vectors

One thorny problem remains, however: when the protocol is executed at the sending server, it needs to decide if a write is new to the receiving server by just comparing the write's accept-stamp with the receiving server's version vector; specifically, the problem occurs when the receiving server's version vector is missing an entry. The sending server has to correctly determine whether the receiver has eliminated the entry because the server corresponding to that entry has retired, or whether the receiver has never heard of the server associated with the missing entry. What writes to propagate critically depends on the outcome of this decision.

More precisely, a server S_i may be absent from another server's version vector for two reasons: either the server never heard about S_i's creation, or it knows that S_i was created and subsequently destroyed. Fortunately, the recursive nature of server identifiers in Bayou allows any server to determine which case holds. Consider the scenario in which R sends S its version vectors during anti-entropy, and R is missing an entry for $S_i = <T_{k,i}, S_k>$. There are two possible cases:

If $R.V(S_k) \geq T_{k,i}$, then server R has seen S_i's creation write; in this case, the absence of S_i from R.V means that R has also seen S_i's retirement. S can safely assume R knows that server S_i is defunct, and does not need to send any new writes accepted by S_i to R.

If $R.V(S_k) < T_{k,i}$, then server R has not yet seen S_i's creation write, and thus cannot have seen the retirement either. S therefore needs to send R all the writes it knows that have been accepted by S_i.

Note that this scenario assumes that R.V includes an entry for S_k. Since multiple servers may retire or be created around the same time, R's version vector may be missing entries for both S_i and S_k in the example used above. Fortunately, the presence of an entry for S_k is not essential to identify retired servers. The solution is based on the recursive nature of the server identifiers. Imagine a CompleteV vector that extends the information stored in the V vector to include timestamp entries for all possible servers. A recursive function can compute entries for this extended vector:

CompleteV$(S_i = <T_{k,i}, S_k>) =$

$V(S_i)$	if explicitly available
plus infinity	if $S_i = 0$, the first server
plus infinity	if $\text{CompleteV}(S_k) \geq T_{k,i}$
minus infinity	if $\text{CompleteV}(S_k) < T_{k,i}$

A value of minus infinity indicates that the server has not yet seen S_i's creation write, and plus infinity indicates that the server has seen both S_i's creation and retirement writes. A server can use the CompleteV function as defined above to always correctly determine which writes to send during anti-entropy.

The dynamic management of version vectors allows a server to create itself by contacting one server for the database. After issuing its creation write, the newly created server needs to perform anti-entropy with the server that just created it. Through this anti-entropy session the newly created replica will itself hold its creation write. Note that the new server's first anti-entropy session is likely to include a full database transfer. After the anti-entropy session between the new server and its creating replica, the creation of a replica can tolerate any other failure, because, by virtue of its server-id and version-vector, the new server holds all the information it needs to positively identify its creation.

5. Discussion

This paper has presented the design of the anti-entropy protocol in a series of steps, each focusing on the features enabled by refinements or extensions of the basic three anti-entropy building blocks: pair-wise communication, exchange of operations, and the ordered propagation of operations. The presentation has been framed in the context of Bayou and its requirements. This section discusses the anti-entropy protocol independently of the Bayou system. It starts with a deconstruction of the protocol, pointing out which of the protocol's properties enable each of anti-entropy's features and how these features can be provided in systems that were designed with some, but not all, of the same components. The section continues by highlighting some of the drawbacks of the design presented in Sections 2-4. Finally, the discussion focuses on some of the policy choices and tradeoffs enabled by the anti-entropy design and the security issues raised by the use of a peer-to-peer model of update propagation.

5.1. Implications for Other Systems

Table 1 presents the dependencies between features of the anti-entropy reconciliation protocol and its design components. Marked entries in the table indicate a dependency between a feature and a design choice. The high-level message of this table is that each of the

Feature \ Design Choices	One-way Peer-to-Peer	Operation-based	Partial Propagation Order	Causal Propagation Order	Stable Log Prefix
Arbitrary Communication Topologies	◆				
Arbitrary Policy Choices	◆				
Low-bandwidth Networks		◆			
Incremental Progress	⬩*	⬩	◆		
Eventual Consistency					◆**
Aggressive Storage Management					◆
Use of Transportable Media	⬩		◆		
Light-weight Dynamic Replica Sets	⬩	⬩		◆	
Per Update Conflict Management		◆			
Session Guarantees				◆	

Table 1: Features enabled by specific anti-entropy design components

* Small marks indicate that the feature is facilitated by the design choice, but does not depend on it.
** Eventual consistency can be supported with the incremental protocol by either establishing a total order on all updates, making operations commutative, or by enforcing a total order on the propagation of updates that are part of the stable prefix.

protocol's features depends only on a few design choices, and that many features can be provided independently.

The unidirectional peer-to-peer reconciliation model supports reconciliation over arbitrary communication topologies and a wide variety of policy choices of when and with which replica to reconcile. Both of these features co-exist independently of the protocol's mechanisms to establish what data to reconcile, the formats used for update propagation, and the order in which data propagates in the system.

The protocol's operation over low-bandwidth networks is enabled through reconciliation based on the propagation of update operations. Low-bandwidth networks can be similarly accommodated by protocols that exchange deltas or differences in the replicas' values. Other systems can reconcile over low-bandwidth networks using either update or delta-based techniques, independently of whether they limit the communication patterns between replicas or whether they impose particular order on the propagation of the updates or deltas.

Update propagation between replicas can make incremental progress if an order can be established over the data to be reconciled. The minimum requirement is a partial order over the updates introduced throughout the system. The peer-to-peer model facilitates the incremental progress of the reconciliation protocol because determining which data needs to be reconciled depends only on the state of two replicas. The incremental nature of the protocol is also facilitated by the propagation of operations or deltas because the unit of propagation is small. Again, incremental progress can be provided independently of the mechanisms provided by the system for other features like data consistency and replica set management.

Anti-entropy's mechanism to cope with the aggressive reclamation of storage resources depends on the stabilization of some prefix of the operations-log. Any mechanism to stabilize the order of operations and to propagate this information back to the replicas is sufficient for this purpose.

Reconciliation using floppy disks or other transportable media can be supported by systems that structure the reconciliation process as a one-way protocol and that provide an ordering over the data to be reconciled. In fact, systems that provide incremental reconciliation protocols are likely to be well suited to also provide file based reconciliation mechanisms.

Light-weight creation and retirement of servers, the last feature introduced for the anti-entropy protocol, depends only on the use of the causal-accept-order defined over updates generated in the system. A system can adopt this technique as long as its reconciliation mechanisms enforce such an ordering during propagation so that version vectors can be used for state representation and operations have a causal relationship. Also, an equivalent of the creation and retirement writes must be provided. However, it is not necessary for these systems to base their reconciliation protocol on operation exchanges per se.

In summary, one of the strengths of the anti-entropy protocol design is that many of its features can be reproduced by other systems, even though they may diverge in the design choices for other functionalities.

5.2. Disadvantages

The variety of features enabled by the anti-entropy protocol result from each replica's ability to reconcile with any other replica in the system, and to do so by only consulting each other's version vectors and the sending server's write-log. The drawbacks of the anti-entropy design are associated exactly with the potential sizes of these two data structures.

The version vectors used for the anti-entropy protocol need an entry for each replica in the system. The size of the vectors thus grows in proportion to the number of replicas and the complexity of the replica creation pattern. When the number of replicas of a database is large compared to the update activity in the system, the cost of exchanging version vectors during an anti-entropy session can become a dominating performance factor. However, as shown in section 6, when servers get created in well-behaved patterns, the version vector size does not have a significant impact on anti-entropy performance for replication factors of a few thousand.

To satisfy Bayou's propagation order requirements, each server must retain all tentative writes in its write-log. Log compaction, such as the removal of entries that first insert and later delete an object from the database, cannot be used. Each server therefore has to keep all the writes it has received until it is notified of the writes' commitment. Hence, if the update activity of the database is large while the commit rate is low, the size of a server's write-log can grow large.

5.3. Anti-entropy Policies

Four types of policies are enabled by the mechanisms of the anti-entropy protocol: policies for when to reconcile, policies for choosing with which replicas to reconcile, policies for deciding how aggressively to truncate the write-log, and policies for selecting a server from which to create a new replica. The different policy choices affect not only the performance and cost of the anti-entropy process but also how quickly updates propagate to other replicas, how up-to-date a replica is, how large the write-log gets, and how quickly writes stabilize.

Potential policies for when to reconcile a replica include: periodic reconciliation, manually triggered reconciliation, and system triggered reconciliation. In other words, a replica can communicate with other replicas at recurring intervals, users can explicitly activate the reconciliation process, and anti-entropy can be initiated by servers when certain system characteristics are met. For example, a server may initiate anti-entropy when a network link becomes available, when it detects the bandwidth is above some threshold, when the CPU load is down, or when its write-log is growing large and committing a series of writes becomes necessary. In general, increasing the frequency of anti-entropy among servers increases the rate of update propagation, the degree to which servers are up-to-date, and the rate at which writes stabilize but at the expense of greater bandwidth consumption due to protocol overheads.

Similarly, policies for choosing with whom to perform anti-entropy can depend on multiple factors: which other replicas are reachable, the network characteristics of the connection to these replicas, the up-to-dateness of the replicas, whether the replicas have truncated too many entries from their logs, and which replica is serving as the primary. Policy choices for anti-entropy partners, like those for when to reconcile, generally trade off bandwidth usage against propagation rates. Previous work on analyzing epidemic-style algorithms, like those used in Bayou, has shown that a judicious choice of anti-entropy partners can dramatically reduce the total traffic required to propagate updates, while yielding only modest increases in the propagation delay [3]. The key is to favor nearby servers and to avoid overloading slow network links. Golding also explored biasing the selection process to favor nearby anti-entropy partners [4].

As discussed in section 3.4, policies to decide how aggressively to truncate the write-log trade off storage and networking resources needed during anti-entropy. Very aggressive write-log truncation may cause lengthy anti-entropy sessions between some servers because of the need to do full database transfers. Observe that write-log truncation policies can be associated with groups of replicas. Within a group, a few replicas may be designated as the servers of choice with whom to reconcile; these replicas could retain more writes in their write-logs to expedite anti-entropy with replicas inside and outside of the group, allowing other replicas in the group to reclaim storage more aggressively.

Finally, policies used to select a server from which to create a new replica can affect the performance of the anti-entropy protocol as shown in section 6. When several servers are available to create a new replica, their respective server identifier lengths should be considered in addition to the other characteristics of these replicas, like up-to-dateness, connection bandwidth, and completeness of their write-logs.

5.4. Security

In addition to the policies discussed thus far, the level of security enforced during the reconciliation process can significantly affect its performance. The peer-to-peer model of the anti-entropy design has a variety of security implications. Replicas may have different levels of trust in other replicas and in clients. The trust relationship may even change depending on the location of a replica, for example, if inside or outside a firewall. The security model may not want to depend on third party authorization services to avoid additional networking requirements. If operations need to be authorized at all servers, security meta-data, like certificates, may need to propagate with the operations. The higher the level of security that is required, the more security related operations need to be executed during the reconciliation process.

The Bayou implementation relies on digital certificates and a hierarchy of trust delegations to implement security. Before untrusted Bayou replicas reconcile, they authenticate each other. In addition, writes include certificates to authorize database accesses for a final time when committed at the primary. Each of these security measures add to the reconciliation times of fully secured Bayou databases.

6. Performance Evaluation

The implementation of the anti-entropy algorithms in Bayou consists of 2846 lines of POSIX-compliant C code. The implementation relies on an existing write-log implementation (1730 lines), a database manager (14768 lines), and utility routines to manipulate version vectors, server identifiers, and write stamps (1081 lines). The implementation also relies on a runtime environment with support for user-level threads, garbage collection, and an RPC package. In this section we present an evaluation of the performance of this implementation, which runs unchanged on both SunOS 4.1.3 and Linux 2.0 platforms.

In summary, the analysis and measurements in this section show that Bayou's anti-entropy protocol performs as expected:

- An anti-entropy session propagates only writes unknown to the receiver, and hence performs as a linear function of the number of such writes and the available network bandwidth;
- While traversing its write-log, the sender spends only a minimal amount of time deciding which writes to propagate;
- The bulk of the anti-entropy algorithm execution time is spent on the network and applying the newly received writes to the write-log and database of the receiver;
- Version vector storage requirements grow between linearly and quadratically with the number of replicas, depending on the pattern in which servers are created from others;
- The simplest implementation of checking whether a write is covered by a version vector takes time between linear and cubic to the number of servers, depending on how the servers are created.

6.1. Experimental Setup

The measurements were taken for BXMH, a version of the popular EXMH e-mail application that uses Bayou instead of a file system to store mail messages. Each experiment measures the time to run the anti-entropy protocol between two replicas of the mail database. In all experiments, only committed writes are propagated, and each write inserts a new e-mail message into the database. In addition, for each anti-entropy session the size of the propagated writes, or e-mail messages, is held constant. Results were collected for two message sizes: 3000 byte messages and 100 byte messages; the message sizes includes both the headers and the message bodies. The large size roughly corresponds to the median size of mail messages. While the small messages were artificially constructed, the large messages correspond to real messages received by one of the authors, either truncated or extended to be 3000 bytes long.

Two platforms were used in the experiments: SPARCstation-10s running SunOS 4.1.3 (labeled SS in the graphs), and 486-based laptops running Linux 2.0 (labeled 486); all machines are clocked at 50MHz. The file system used to store the replica's write-log on the SPARCstations is NFS, whereas the laptops use UFS. Two types of network connections link the replicas in these experiments: a one hop, 10 Mbps ethernet connection, and a PPP (point-to-point protocol) connection over a 9.6 Kbps modem and then two ethernet hops to the second server within a firewall. In practice, the achievable bandwidth over the modem reaches up to 26.2 Kbps due to compression.

Each measurement was taken at least five times, for the faster experiments sometimes up to ten times. All figures present the averages over all runs. They include error bars if these do not clutter the presentation, otherwise standard deviations are reported in the captions.

6.2. Anti-entropy Execution Times

Figure 5 demonstrates that the execution time of the anti-entropy protocol is a linear function of the number of new writes being propagated. The slope of the function depends on the size of the messages being exchanged and the network bandwidth available for reconciliation.

The range of performance observed for anti-entropy of 3000-byte e-mail message writes starts with two servers running on the SPARCstations communicating over the ethernet, which requires 2.26 seconds to propagate the first write and a little under 5 seconds for each additional 100. At the other extreme, for the laptop and modem, it takes 7.16 seconds for the first write and about 150 seconds for each additional 100 writes.

These numbers have significant room for improvement since each Bayou write that adds an e-mail message to the BXMH database currently has two substantial data

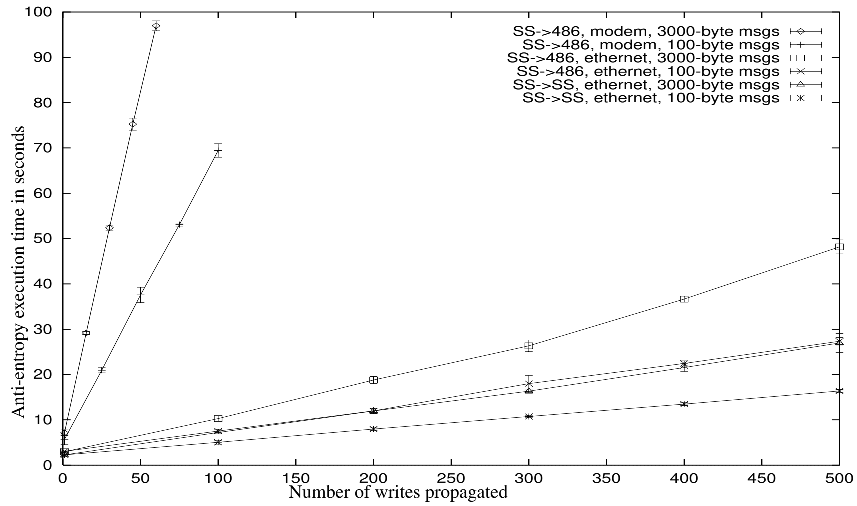

Figure 5. Anti-entropy execution as a function of the number of writes propagated
(each write corresponds to one mail message)

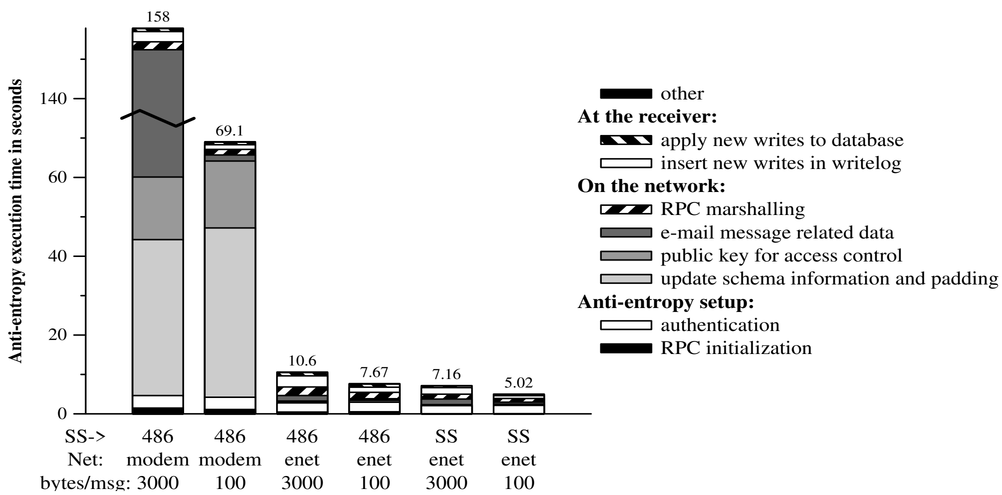

Figure 6. Anti-entropy execution time breakdown for the propagation of 100 writes
(standard deviations on all total times are within 2.2% of the reported numbers)

overheads: 520 bytes for the public key of the principal that is updating the database, which is included for access control purposes; and 1316 bytes of update schema information and data cell padding. These update schema and cell padding overheads are unnecessarily large; 40% of the overhead corresponds to the ASCII strings of the column names of each field being updated, while the remaining fraction of the 1316 bytes are zero filled to pad data cells because SunRPC represents everything by an integral number of 4-byte words. As shown below, more sophisticated representations for update schema and cell padding would substantially improve the performance of anti-entropy over the modem. Similarly, systems with different access control policies or mechanisms could obviate the need to transmit public keys with every write and, with that, also reduce the overall communication overhead.

To further analyze the performance of the algorithm, we broke down the execution time of anti-entropy sessions that propagate 100 writes between two replicas. Each of the bars in Figure 6 corresponds to a different experimental configuration. The labels indicate on which platform the receiving replica ran, what network connection was used, and which size messages were propagated. In all cases the sending replica ran on a SPARCstation, which does not affect the performance of anti-entropy since, as the cost breakdowns show, the overheads at the sending replica are minimal.

As Figure 6 shows, the factors that contribute the most to the performance of the anti-entropy interactions are:

- **Network transfer**: the most significant overhead of the anti-entropy sessions corresponds to the actual transmission of the writes. This phase includes the marshalling and unmarshalling of the writes being

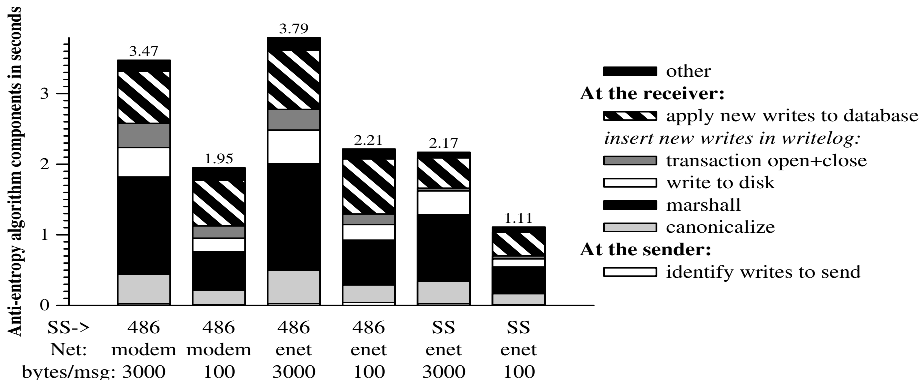

Figure 7. Network independent anti-entropy algorithm components for the propagation of 100 writes
(standard deviations on all total times are within 2.9% of the reported numbers)

propagated, as well as the time on the network itself. In the graph, the network transfer time is subdivided into four categories: time to marshall and unmarshall the RPC data, time related to transfer actual message information, time to transmit each write's public key and the overhead to transmit the update schema information and data padding for each write. As mentioned earlier, systems with different access control mechanisms and more efficient schema and padding implementations could eliminate a substantial part of the communication overhead, particularly in the modem cases.

The figure also shows that the bandwidth over the ethernet between the SPARCstations is about double that achieved between the laptop and the SPARCStation; the bandwidth in the former case varied between 4.6 and 5.8Mbps, while the laptop's ethernet connection only achieved 2.4-3.1Mbps. The observed bandwidth over the modem varied between 23.3 and 26.2Kbps for the communication of the two sets of 100 writes.

- **Anti-entropy setup**: during this step the sender locates other replicas using the name service, sets up the RPC handle to communicate with the receiving replica, and performs the challenge response protocol that is used in Bayou to mutually authenticate replicas before they engage in the actual propagation of writes. The last response of the authentication protocol also includes the version vector state information from the receiving replica. This setup time accounts for most of the time it takes to transmit one write. For 3000 byte messages the setup time in the ethernet case is 2.08 seconds and in the modem case 4.63 seconds, which corresponds to 65-88% of the first write propagation times reported earlier.
- **Applying the newly received writes at the receiver**: the last observable overhead in Figure 6 is due to the processing of the received writes at the receiving replica, both incorporating the writes into the write-log and applying them to the database.

Since the network transfer time dwarfs most of the other execution time components of anti-entropy, Figure 7 shows the smaller overheads of the algorithm; they exclude the network transfer time and the anti-entropy setup time. We chose not to include the setup time because it involves operations such as authentication, which other implementations may choose to exclude, and RPC initialization, which can be implemented differently by other systems. These smaller components also correspond to the network independent portion of the anti-entropy algorithm.

Of these network independent overheads, the largest component corresponds to inserting all newly received writes in the receiver's write-log. Figure 7 breaks that operation down into four subcomponents: canonicalizing of the new writes for string sharing purposes, marshalling the new writes to serially write the in-memory data structures out to disk, the actual disk I/O time, and the transactional overhead to manipulate the write-log. Bayou's current implementation is based on an in-memory database and string canonicalization is necessary to reduce memory overheads, but may not be needed in other systems. The marshalling implementation is not optimized either. We believe that the canonicalizing and marshalling steps have ample opportunity for performance improvements.

The second largest cost component shown in Figure 7 corresponds to applying the 100 newly received writes to the receiver's database. This time ranges between 335 msec. on the SPARCStation and 959 msec. on the laptop. Database performance will necessarily vary depending of the database manager used in the system's implementation. These numbers should therefore only be considered as one performance example.

Finally, Figure 7 shows that the time spent in the algorithm to determine which writes are new to the receiving replica is negligible, taking as little as 11 msec. and only a maximum of 42 msec. in these experiments, in which all writes are unknown to the receiver. Separately

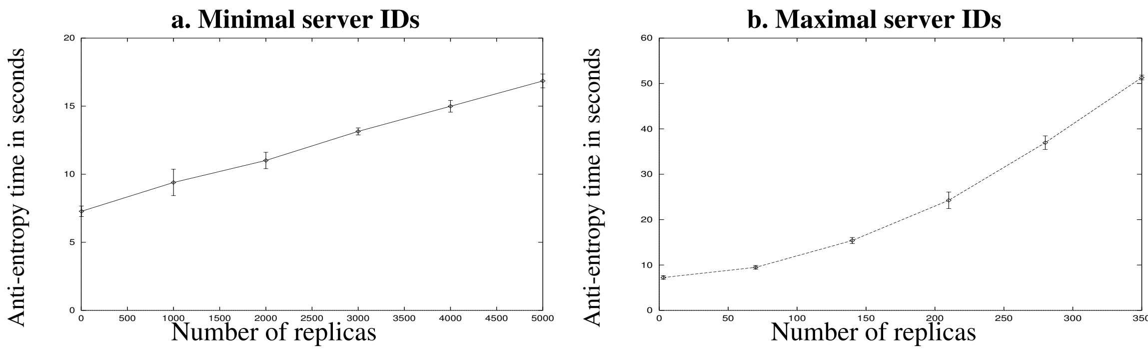

Figure 8. Anti-entropy execution time for 100 writes as a function of the number of replicas

we also measured the time for a sending server to traverse 100 entries of its write-log when all of these writes are already known to the receiver. For all the server pairs, network combinations, and write sizes reported, this time is less than a tenth of a second.

6.3. Effect of Server Creation Patterns

As mentioned in section 4.3, the space required to represent a version vector depends on the pattern of server creations. To be concrete, the representation used "on the wire" in our RPC protocol is the SunRPC representation of a sequence of <server-id, accept-stamp> pairs. This representation takes

$$4 + 12N + \sum_{i=1}^{N} 8|Si|$$

bytes for a vector of N servers, where |Si| is 0 for the initial server of a system and 1 + |Sk| when Si is created from server Sk. Accept-stamps are 8 bytes long. In the most storage efficient case, all servers are created from the initial one, and the version vector representation requires 20N - 4 bytes, or 20 Kilobytes for 1000 servers. In the least storage efficient case, servers are created in one long chain, and the version vector representation requires 4N^2 + 8N + 4 bytes, or 4 Megabytes for 1000 servers.

The time to test whether a given write is among those represented by a version vector may, in the worst case, grow cubically. This time analysis is based on a very simple implementation that uses a list data structure for the version vector representation. The three multiplicative factors result from: (1) traversing the list representation of the version vector, (2) comparing the server identifiers of each entry in the list with the server-id of the write, and (3) if an entry for a server is not present, recursively calculating CompleteV. More sophisticated data structures like hash tables with a one way hash function could be used instead, making the time linear.

Figure 8.a shows that the execution time of anti-entropy is a linear function of the number of servers in the system for the least costly server creation pattern. It also shows that 1000 servers can be supported easily. On the other hand, as shown in Figure 8.b, execution times grow quadratically in the most costly server creation pattern. The cubic factor does not appear in Figure 8.b because no servers had been retired from the system.

The measurements in Figure 8 correspond to the propagation of 100, 3000-byte messages between SPARCstations over the ethernet connection. The measured execution times include both the writes' accept stamps to version vector comparisons and the marshalling and network time for the initial back-flow of the receiver's version vector. In comparison, the numbers reported in section 6.2 correspond to a system with three replicas.

If the size of version vectors causes performance problems due to exceedingly long server-identifiers, the lightweight server creation and retirement can be used to institute practical policies to limit the number and lifetime of servers whose creation was far removed from the initial server. For example, a server can recreate itself with a smaller identifier if it locates another server with a smaller identifier than the one of the server's original creator. This process requires the issuance of a retirement write, a new creation write, and anti-entropy with the new creator; it can be optimized to reuse the server's existing write-log and database.

7. Related Work

A number of research and commercial systems have used weak consistency replication and propagated updates among replicas in a lazy fashion. Each of the individual features of Bayou's anti-entropy protocol have almost certainly appeared in previous systems in some form. Interesting differences lie in the implementation details about what information gets exchanged between replicas, what data structures are used to keep track of other replicas and the state of these replicas, what communication patterns are allowed between replicas, and so on. Unfortunately, detailed information about how other systems reconcile their replicas is difficult to obtain, especially for commercial products. One contribution of this paper is describing a reconciliation protocol in detail along with the design decisions that went into it. In this section, we discuss how other systems' protocols compare to Bayou's based on the sketchy information available.

Grapevine, one of the earliest weakly replicated systems, propagated updates via electronic mail [2]. Electronic mail is not completely reliable, however, so the product version of Grapevine, called Clearinghouse [15], added a background anti-entropy process in addition to mail delivery. It was later realized that epidemic style algorithms, like Clearinghouse's anti-entropy, could be used by themselves to fully propagate updates [3]. Pair-wise reconciliation of replicas is currently used in several systems besides Bayou, including Notes [10], Ficus [7], and refdbms, which uses Golding's timestamped anti-entropy protocol [4].

Rather than a peer-to-peer model in which any replica can contact any other replica to reconcile their data, some systems organize replicas into a hierarchy where a replica only exchanges updates with its parent or children. Examples of this are the client-server reconciliation protocols in file systems like Coda [11] and distributed object systems like Rover [9], and also the primary-secondary or master-snapshot protocols in database management systems like Oracle [16] and Sybase [5]. Due to their simplified communication patterns, these systems can more easily maintain accurate information about the state of the replica(s) with which they exchange updates. However, update propagation is more affected by communication outages.

In many systems using lazy replication, the information exchanged between replicas is based on data objects with associated update timestamps or version vectors. This is true for Grapevine [2], Clearinghouse [15], Notes [10], and Microsoft Access [8]. File systems like Coda [18] and Ficus [7] exchange updated files between servers or between clients and servers. The notion of reconciling logs of update operations held at various replicas, as is done in Bayou via the anti-entropy protocol, has been discussed for some time in the literature [1, 17, 21] and is used in some commercial database systems [5, 16]. Oracle7, for instance, uses asynchronous RPCs to propagate transactions between a master and its snapshots or other masters [16]; it does not, however, allow these transaction to propagate through intermediary servers. Rover also uses operations as the unit of reconciliation by queuing RPC invocations that are eventually applied to the master copy of an object [9].

Systems that propagate updated data objects need an additional mechanism to handle deleted objects. For example, Clearinghouse servers maintained and exchanged “death certificates” for deleted objects [3, 15]. Protocols have been devised to decide when replicas can safely discard deleted data [17]. When update operations rather than data are used for reconciliation, deletions are handled automatically as just another type of update operation, and servers can immediately reclaim the space used by deleted data items.

A goal in the design of Bayou's anti-entropy protocol was to ensure that servers can make progress even if the protocol is disrupted by the loss of a network connection. That is, a server should be able to use and propagate to other replicas any updates that it receives even if the protocol does not complete successfully. Some systems run their reconciliation process as an atomic transaction and hence lose the incremental property. Coda has added a trickle reintegration protocol for use by weakly-connected clients; while this protocol is atomic, it includes the notion of a chunk size that can be set to a small value to achieve incremental reintegration [14]. Also note that systems based on queued RPCs, such as Oracle, can make incremental progress since each RPC is generally run as a separate transaction [16].

Techniques for changing the set of replicas vary widely among systems. In systems with a client-server or primary-secondary relationship between replicas, new clients or secondaries can generally be created by simply contacting the primary site. In peer-to-peer systems, adding or removing replicas often requires a system administrator and reconciliation between replicas. Golding uses a group membership protocol that requires a new replica to find some number of sponsor replicas and a retiring replica to wait until notice of its retirement reaches all other replicas [4]. Notes [10] and Microsoft Access [8], as far as we can tell, are like Bayou in that they allow replicas to be created readily from any existing replica, though it does not appear that the knowledge of new replicas is propagated throughout the system as in Bayou. Bayou is the first system we know about that employs version vectors to characterize a replica's contents, allows any replica to accept updates, and yet permits lightweight creation and retirement of replicas.

8. Conclusions

The major contribution of this paper is in the detailed presentation of Bayou’s protocol for lazily propagating updates between its weakly consistent replicas along with the rationale for each design decision and the features enabled by it. The protocol is practical, implemented, and quite simple. Three basic design decisions went into Bayou’s anti-entropy protocol: the model of pair-wise reconciliation between peer replicas, the exchange of write operations stored in per-replica logs that are compactly characterized using version vectors, and the propagation of writes between replicas in an order that is closed with respect to the writes’ accept, causal, or total order.

Although none of these design decisions in isolation is particularly novel, together they provide the flexibility necessary to cope with the diversity of networking environments in common use today, including unreliable wireless networks, dial-up modems over telephone lines, the global Internet, and even "sneakernet". They permit replicas to make incremental progress towards their convergence in the face of involuntary disconnections while giving replicas control over the pruning of their individual write-logs. They also support Bayou's style of conflict resolution and its session guarantees.

Additionally, Bayou incorporates a new lightweight mechanism for creating and retiring replicas that builds on and is compatible with its anti-entropy protocol. Special creation and retirement writes propagated via anti-entropy and server identifiers built from a hierarchy of write-stamps permit replicas to reconcile any differences that may exist in their views of the current replica set.

Key to the flexibility of Bayou's anti-entropy design is the separation from the protocol itself of the policies for choosing pairs of replicas to reconcile and at what times. Optimal policies for choosing anti-entropy partners depend on a number of complex factors such as the available bandwidth between replicas and the cost, perhaps in real money, of communication between them. Admittedly, the current Bayou system uses the most simple policies imaginable, like random selection. Exploring different policies and their effect on the rate and cost of overall system convergence, as well as on the total storage requirements, remains an area for fertile research.

9. Acknowledgments

Carl Hauser and Brent Welch participated in the design of the anti-entropy protocol. Susan Owicki, again, provided invaluable help with the final revisions to this paper. Finally, we thank the anonymous referees for their thorough comments and suggestions.

10. References

[1] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *The Computer Journal* 34(6):534-541, December 1991.

[2] 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.

[3] 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.

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

[5] A. Gorelik, Y. Wang, and M. Deppe. Sybase Replication Server. *Proceedings 1994 ACM SIGMOD Conference*, Minneapolis, Minnesota, May 1994, page 469.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings 1996 ACM SIGMOD Conference*, Montreal, Canada, June 1996, pages 173-182.

[7] 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.

[8] B. Hammond. WINGMAN: A replication service for Microsoft Access and Visual Basic. Microsoft white paper.

[9] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. *Proceedings Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995, pages 156-171.

[10] 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.

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

[12] 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.

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

[14] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *Proceedings Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995, pages 143-155.

[15] D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems* 1(3):230-253, July 1983.

[16] Oracle Corporation. *Oracle7 Server Distributed Systems: Replicated Data, Release 7.1*. Part No. A21903-2, 1995.

[17] S. Sarin and N. A. Lynch, Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering* SE-13(1):39-47, January 1987.

[18] 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.

[19] 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.

[20] 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. *Proceedings Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995, pages 172-183.

[21] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problem. *Proceedings Third ACM Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, August 1984, pages 233-242.