Flexible Update Propagation for Weakly Consistent Replication

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

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.

Anti-Entropy Algorithm

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, and
3. write propagation is constrained by the accept-order.

Anti-Entropy Algorithm

```
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)

Anti-Entropy Algorithm

```
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
}
```

basic

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

Anti-Entropy Algorithm

```
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
}
```

committed writes

basic

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

Anti-Entropy Algorithm

```
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
}
```

write-log truncation

committed writes

basic

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

Anti-Entropy Algorithm

```
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)

Creation Writes

- Server S_i creates itself by sending a creation write to another server S_k, which handles it like any client write
- Write: <infinity, $T_{k, i}$, S_k>
- Entry for S_i added to version vectors
- S_i's server-id: <$T_{k, i}$, S_k>
- S_i initializes accept-stamp counter with $T_{k, i}$+1

Creation Writes

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.

Retirement Writes

- When server wants to die, it issues retirement write to itself (also like any other write), stops accepting client writes
- Must stay alive until it performs anti-entropy with ≥ 1 other server
- When server receives retirement write, updates version vectors

Logically Complete Version Vectors

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.

Features

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.

Disadvantages

- 1 vector for each replica – inefficient when number of replicas > update activity
- Must retain all tentative writes until commit – inefficient when update activity > commit rate

Anti-Entropy Policies

- When to reconcile
- With which replicas to reconcile
- How aggressively to truncate the write-log
- Selecting a server from which to create a new replica

Performance Evaluation

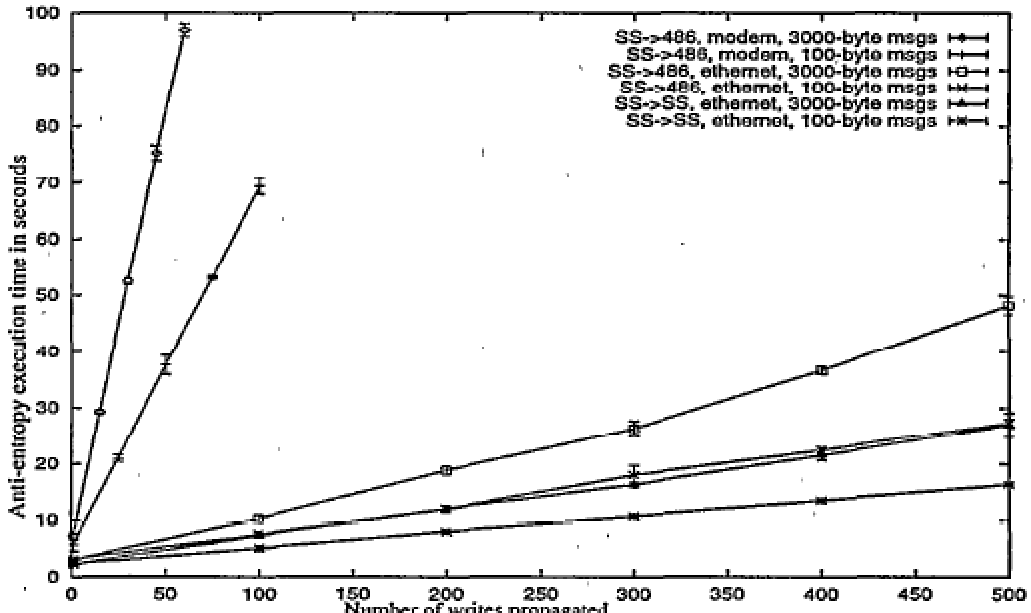

Figure 5. Anti-entropy execution as a function of the number of writes propagated
(each write corresponds to one mail message)

Performance Evaluation

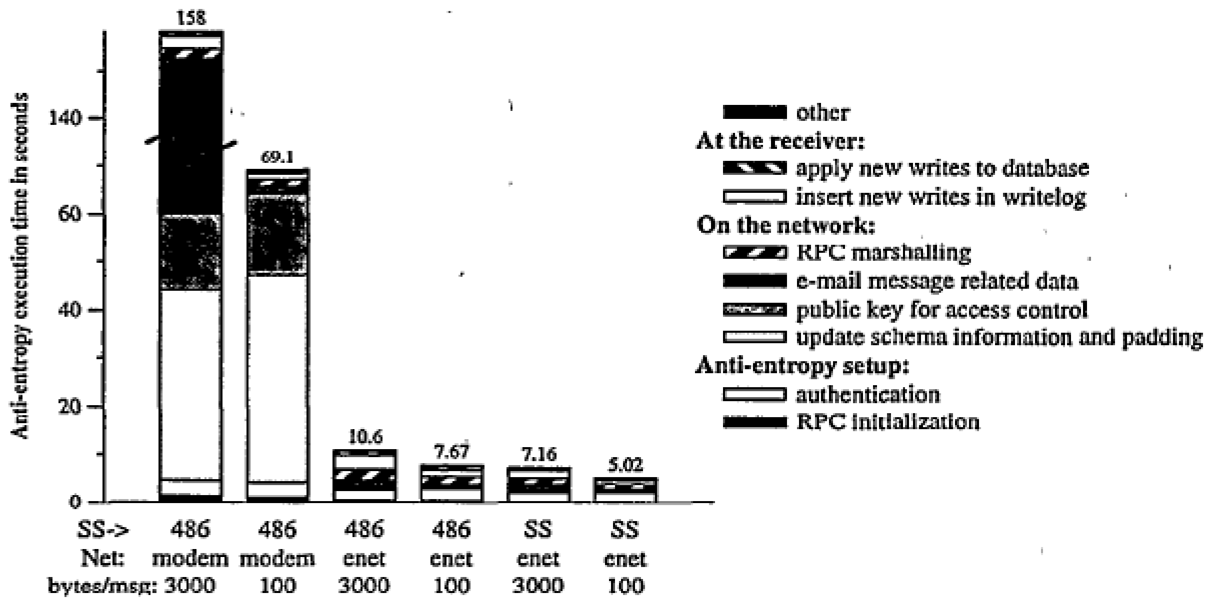

Figure 6. Anti-entropy execution time breakdown for the propagation of 100 writes
(standard deviations on all total times are within 2.2% of the reported numbers)

Performance Evaluation

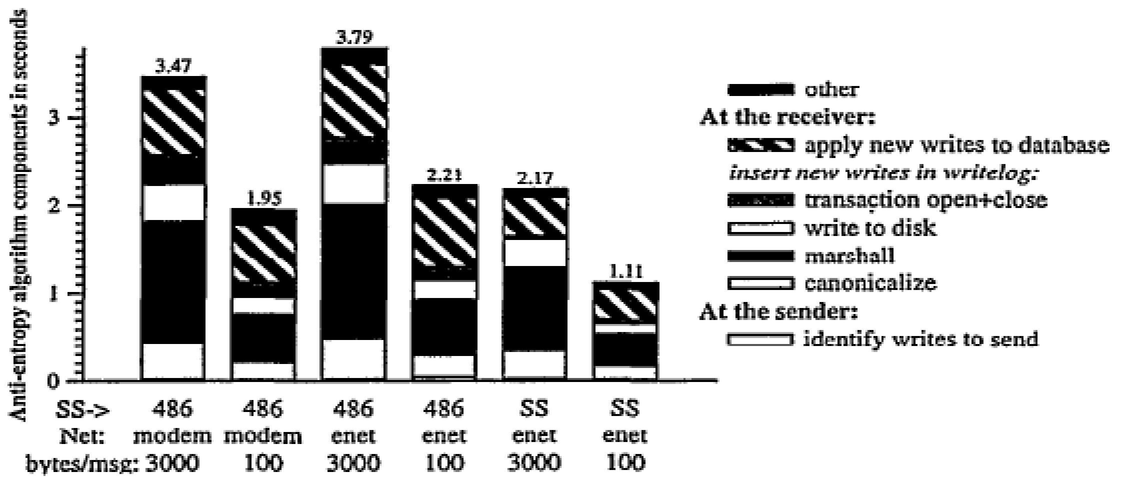

Figure 7. Network independent anti-entropy algorithm components for the propagation of 100 writes
(standard deviations on all total times are within 2.9% of the reported numbers)

Performance Evaluation

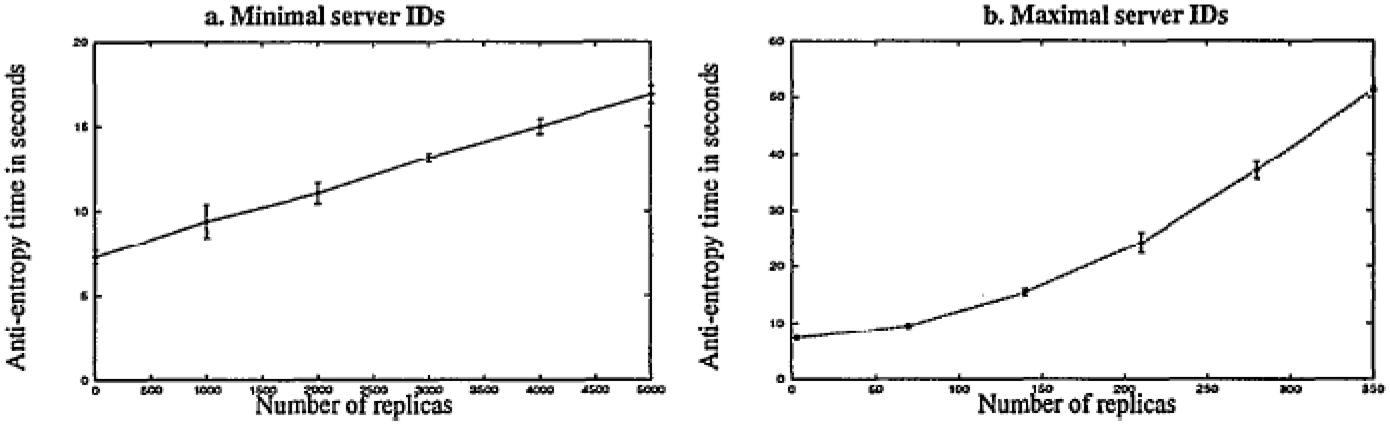

Figure 8. Anti-entropy execution time for 100 writes as a function of the number of replicas