Scalable Update Propagation in Epidemic Replicated Databases

Michael Rabinovich, Narain Gehani, and Alex Kononov

AT&T Bell Laboratories
600 Mountain Ave,
Murray Hill, NJ 07974

Abstract. Many distributed databases use an *epidemic* approach to manage replicated data. In this approach, user operations are executed on a single replica. Asynchronously, a separate activity performs periodic pair-wise comparison of data item copies to detect and bring up to date obsolete copies. The overhead due to comparison of data copies grows linearly with the number of data items in the database, which limits the scalability of the system.

We propose an epidemic protocol whose overhead is linear in the number of data items being copied during update propagation. Since this number is typically much smaller than the total number of data items in the database, our protocol promises significant reduction of overhead.

1 Introduction

Data replication is often used in distributed systems to improve system availability and performance. Examples of replicated systems abound and include both research prototypes (e.g., [5, 14]) and commercial systems (e.g., [8, 10]).

Many of these systems use an *epidemic* [4] approach to maintain replica consistency. In this approach, user operations are performed on a single replica. Asynchronously, a separate activity (termed *anti-entropy* in [4]) compares version information (e.g., timestamps) of different copies of data items and propagates updates to older replicas.

Epidemic protocols exhibit several desirable properties: user requests are serviced by a single (and often a nearby) server; update propagation can be done at a convenient time (i.e., during the next dial-up session); multiple updates can often be bundled together and propagated in a single transfer.

A significant problem with existing epidemic protocols is the overhead imposed by anti-entropy. This overhead includes periodic pair-wise comparison of version information of data item copies to decide which copy is more recent. It therefore grows linearly with the number of data items in the system. This limits the scale the system can achieve without significant performance degradation.

It might appear that a simple solution to this problem exists where each server would accumulate its updates and periodically push them to all other replicas, without any replica comparison. However, the following dilemma arises. If recipients of the updates do not forward them further to other nodes, then full responsibility for update propagation lies with the originating server. A failure of this server during update propagation may leave some servers in an obsolete state for a long time, until the originating server is repaired and completes the propagation. On the other hand, forwarding updates by servers to each other would create a lot of redundant traffic on the network.

In this paper, we propose a different solution to the anti-entropy overhead problem. We present a protocol that, like existing epidemic protocols, performs periodic comparison of version information of replicas to determine which replicas are out-of-date. Unlike existing epidemic protocol, our protocol detects whether update propagation between two replicas of the *whole database* is needed in constant time, independently of the number of data items in the database. Moreover, when update propagation is required, it is done in time that is linear in the number of data items to be copied, without comparing replicas of every data item. Typically, the number of data items that are frequently updated (and hence need to be copied during update propagation) is much less than the total number of data items in the database. Thus, our protocol promises significant reduction of overhead.

Our protocol is based on *version vectors*, first proposed in [12] to detect inconsistencies between replicas of a data item and widely used for various purposes in distributed systems. In existing replicated systems based on version vectors, a server i associates a *version vector* with every data item replica x_i stored on this server. This vector (described formally later in the paper) records in its j-th component the number of updates originally performed by server j and reflected in x_i. By comparing the version vectors of two copies of a data item, the system can tell which of the two is more recent. Therefore, the older replica can catch up, e.g., by copying the newer replica. As other existing epidemic protocols, version vector-based protocols impose significant overhead on the system due to pair-wise comparison of version information (version vectors in this case) of data item replicas.

The initial idea behind our protocol is simple: associate version vectors with the entire database replicas, instead of (or, in fact, in addition to) replicas of individual data items. Then, perform anti-entropy between replicas of entire databases. Anti-entropy would compare the version vectors of two database replicas to detect in constant time whether update propagation between replicas of *any* data item in the database is required, and if so, infer which data items must be copied by looking at these version vectors and database logs.

There are two main challenges in implementing this idea. First, our goal to limit update propagation time dictates that only a constant number of log records per data item being copied can be examined or sent over the network. However, the number of log records is normally equal to the number of updates and can be very large. This problem required an interesting mechanism for log management.

The second problem is due to the mismatch between managing version vectors at the granularity of entire database replicas and maintaining replica consistency at the granularity of individual data items. Specifically, our idea implies that update propagation is always scheduled for all data items in the database at once. In contrast, existing version vector-based protocols allow update propagation to be scheduled differently for each individual data item. While it is not feasible to provide different schedules for *each* data item in the database, the ability to, say, reduce the update propagation time for some key data items is important. Thus, we must allow nodes to obtain a newer version of a particular data item at any time, in addition to normally scheduled update propagation. We call these data items, obtained by direct copying outside the normal update propagation procedure, *out-of-bound* data items. Out-of-bound data violates certain ordering properties of updates on a node, which are essential for the correctness of our protocol.

We deal with this problem by treating out-of-bound data completely differently from the rest of the data. When a node copies a data item out of bound, it does not modify its database version vector or database logs. Instead, it creates parallel data structures: an *auxiliary* data item and an auxiliary log. The node then uses the auxiliary copy for servicing user operations and requests for out-of-bound copying of this data item from other nodes. At the same time, the node uses the "regular" copy of the data item for scheduled update propagation activity. In addition, a special *intra-node update propagation* procedure ensures that updates from the auxiliary log are eventually applied to the regular copy without violating the update orderings required by the protocol. When the regular copy catches up with the auxiliary copy, the latter is discarded.

This separation of out-of-bound and "regular" data is achieved at the expense of additional costs, both in storage for keeping auxiliary data and in processing time for intra-node update propagation. Thus, the assumption behind our protocol is that the number of out-of-bound data items is small relative to the total number of data items.

A secondary contribution of this paper is that it explicitly specifies correctness criteria for update propagation, separately from correctness criteria for database transactions. This allows formal reasoning about correct propagation independently from the data consistency guarantees provided by the system.

2 The System Model

We assume a collection of networked servers that keep databases, which are collections of data items. A database can be replicated (as a whole) on multiple servers. We will refer to an instance of a database (data item) kept on an individual server as a *replica* or *copy* of the database (data item). For simplicity, we will assume that there is a single database in the system. When the system maintains multiple databases, a separate instance of the protocol runs for each database.

Different replicas of a data item are kept consistent using an epidemic protocol: user operations are serviced by a single server; asynchronously, updates are propagated throughout the network by an anti-entropy process. In addition to periodically scheduled update propagation, a server may obtain a newer replica of a particular data item at any time (*out-of-bound*), for example, on demand from the user.

Update propagation can be done by either copying the entire data item, or by obtaining and applying log records for missing updates. For instance, among commercial systems, Lotus Notes uses whole data item copying, while Oracle Symmetric Replication copies update records. The ideas described in this paper are applicable for both these methods. We chose whole data copying as the presentation context in this paper.

The protocol is targeted towards applications where the number of data items copied during update propagation is small compared to the total number of data items. In other words, the fraction of data items updated on a database replica between consecutive update propagations is in general small. Another assumption for the workload is that relatively few data items are copied out-of-bound.

We do not make any assumptions about the level of replica consistency guaranteed by the system. The system may enforce strict consistency, e.g., by using tokens to prevent conflicting updates to multiple replicas. (In this approach, there is a unique token associated with every data item, and a replica is required to acquire a token before performing any updates.) Or, the system may use an optimistic approach and allow any replica to perform updates with no restrictions. In the latter approach, when conflicting updates are discovered, they are resolved in an application-specific manner (which often involves manual intervention).[1]

Neither do we assume anything about the transactional model supported by the system. The system may use two-phase locking [2] on an individual server while relying on optimism for replica consistency. The system can also choose to provide guaranteed serializability of transactions by executing on top of a pessimistic replica control protocol. (See [2] for the serializability theory in replicated systems.) Finally, the system may not support any notion of multi-data-item transactions at all (like Lotus Notes, [9]).

Finally, to simplify the presentation, we assume that the set of servers across which a database is replicated is fixed.

2.1 Correctness Criteria

We assume that actions performed by individual nodes are atomic. In particular, any two updates or an update and a read on the same data item replica are executed in some serial order. Thus, all updates reflected in a replica form a serial *history* $h = \{op_1, \ldots, op_n\}$, according to the order in which these updates executed.

To reason about correctness of update propagation, we need a few definitions.

Definition 1 Inconsistent data item replicas. Let x_A and x_B be two replicas of a data item x. x_A and x_B are called *inconsistent* or *in conflict* if there are updates op_i and op_j such that x_A reflects update op_i but not op_j, while x_B reflects op_j but not op_i.

Definition 2 Older and newer data item replicas. x_A is called *older* or *less recent* than x_B (and x_B is called *newer* or *more recent* than x_A) if the update history of x_A is a proper prefix of the update history x_B. A replica of a data item is called *obsolete* if there is a newer replica of the same data item in the system.

We assume the following correctness criteria for update propagation.

1. Inconsistent replicas of a data item must be eventually detected.
2. Update propagation cannot introduce new inconsistency. In other words, data item replica x_i should acquire updates from x_j only if x_j is a newer replica.
3. Any obsolete data item replica will eventually acquire updates from a newer replica. In particular, if update activity stops, all data item replicas will eventually catch up with the newest replica.

[1] We do not get into the discussion of tradeoffs between optimistic and pessimistic replica management. The ideas we present here are equally applicable to both approaches.

3 Background: Version Vectors

Version vectors were proposed in [12] to detect inconsistency among replicas in distributed systems, and have been widely used for various purposes in distributed systems. We describe some existing applications of version vectors in the review of related work (Section 8).

Consider a set of servers, $\{1, \ldots, n\}$, that keep copies of a data item x. Denote x_i to be the copy of x kept by server i. Every server i maintains a *version vector* $v_i(x)$ associated with its copy of x. This version vector has an entry (an integer number) $v_{ij}(x)$ for each server j that keeps a copy of the data item.

The rules for maintaining version vectors are as follows. Upon initialization, every component of the version vector of every replica of the data item is 0.

When a server i performs an updates of data item x_i, it increments its "own" entry (i.e., $v_{ii}(x)$) in its version vector for x_i.

When server i obtains missing updates for x from a server j (either by copying the whole data item or by obtaining log records for missing updates), i modifies $v_i(x)$ by taking the component-wise maximum of $v_i(x)$ and $v_j(x)$: $v_{ik}^{new}(x) = \max(v_{ik}^{old}, v_{jk})$, $(1 \leq k \leq n)$.

The following fact about version vectors has been shown [11].

Theorem 3. *At any time, $v_{ij}(x) = u$ if and only if i's replica of x reflects the first u updates that were made to this data item on server j.*

In particular, these corollaries hold:

1. If two copies of the same data item have component-wise identical version vectors, then these copies are identical.
2. For two replicas of the data item, x_i and x_j and some server k, let $v_{ik} < v_{jk}$ and $v_{jk} - v_{ik} = u$. Then, x_i has seen u fewer updates performed on server k and reflected on x_j. Moreover, these missing updates are the last updates from server k that were applied to x_j.
3. A copy x_i is older than a copy x_j iff (a) version vector $v_i(x)$ is component-wise smaller or equal to $v_j(x)$, and (b) at least one component of $v_i(x)$ is strictly less than the corresponding component of $v_j(x)$. ($v_j(x)$ is said to *dominate* $v_i(x)$ in this case).
4. Copies x_i and x_j are inconsistent iff there exist k and l such that $v_{ik}(x) < v_{jk}(x)$ and $v_{il}(x) > v_{jl}(x)$. We will call two version vectors with this property *inconsistent* version vectors.
   Indeed, this means that x_i has seen some updates (made on server l) that x_j has not received; at the same time, x_i has not seen some updates made on server k and reflected in x_j.

Given these facts, update propagation can be done by periodically comparing version vectors of pairs of data item replicas and either doing nothing (if both replicas have identical version vectors), or bringing the older replica up-to-date, or flagging a conflict.

4 Data Structures

This section describes the data structures and some utility functions used by the protocol. This is followed by the description of the protocol in the next section.

Consider a system with n nodes replicating a database. As before, every node i maintains an item version vector (IVV) for every data item x_i in i's replica of the database. Additional data structures are described in the rest of this section.

4.1 Database Version Vectors

Our protocols associate version vectors with entire replicas of databases. These version vectors are referred to as *database version vectors*, or *DBVV*, as opposed to data item version vectors, or *IVV*, described in Section 3. DBVV is similar in many ways to IVV, except its components record the *total* number of updates performed on corresponding servers to *all* data items in the database replica.

More formally, node i keeps a database version vector V_i with n components, where n is the number of nodes maintaining a replica of the database, with the following maintenance rules.

1. Initially, all components of V_i are 0.
2. When node i performs an update to any data item in the database, it increments its component in the database version vector: $V_{ii}^{new} = V_{ii}^{old} + 1$.
3. When a data item x is copied by i from another node j, i's DBVV is modified to reflect the extra updates seen by the newly obtained copy of the data item: $V_{il}^{new} = V_{il}^{old} + (v_{jl}(x) - v_{il}(x))$, $1 \leq l \leq n$, where $v_{km}(x)$ is the mth component of the IVV associated with data item x on node k.

To get an intuition behind the last rule, consider an arbitrary node l. x_i has seen $v_{il}(x)$ updates originally performed by l, and x_j has seen $v_{jl}(x)$ such updates. Our protocol will copy x from j to i only if x_j is more recent. Thus, $v_{il}(x) \leq v_{jl}(x)$, and the additional number of updates seen by x_j that were originated on l is $(v_{jl}(x) - v_{il}(x))$. Once i copies the value of x_j, the total number of updates originated on l and seen by all data items in i's database increases by this amount. Therefore, component V_{il} of ith DBVV must increase accordingly.

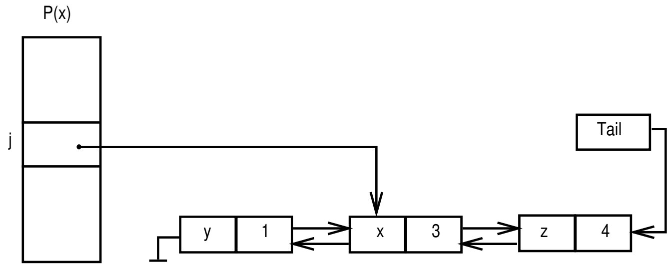
4.2 The Log Vector

Node i maintains a *log vector* of updates, L_i. Each component, L_{ij}, records updates performed by node j (to any data item in the database) that are reflected on node i. The order of the records in L_{ij} is the same as the order in which j performed the updates.

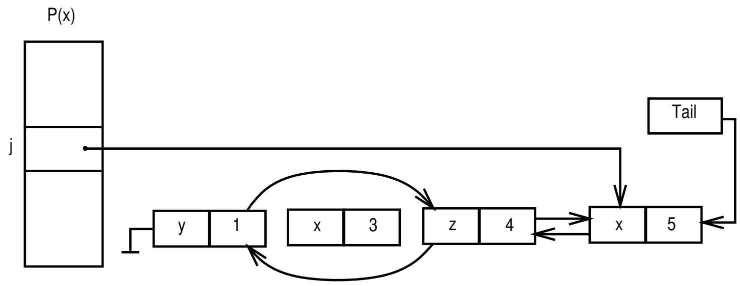
Records are added to the log when node i performs updates to non out-of-bound data items. New records can also be added to the log when they are obtained from the source node during update propagation.

A log record has a form (x, m), where x is the name of the updated data item, and m is the value of V_{jj} that node j had at the time of the update (including this update). Recall from Section 4.1 that V_{jj} counts updates performed by j. So, m gives the sequence number of the update on node j. Note that log records only register the fact that a data item was updated, and not information to re-do the update. Thus, these records are very short.

The key point is that, from all updates performed by j to a given data item that i knows about, only the record about the *latest* update to this data item is retained in L_{ij}. Hence, when a new record (x, m) is added to L_{ij}, the existing record referring to the same data item is discarded.



(a) The structure of log component L_{ij}.



(b) The structure of L_{ij} after adding record (x,5).

Fig. 1. The structure of a log component.

To do this efficiently, all records in L_{ij} are organized in a doubly linked list (see Figure 1). An array of pointers $P(x)$ is associated with every data item x_i. Its component P_j contains the pointer to the existing record in L_{ij} referring to x. When a new record (x, m) is added, the following *AddLogRecord* procedure is executed:

AddLogRecord(node number j, record $e = (x, m)$):
  The new record e is linked to the end of log L_{ij};
  The old log record referring to the same data item is located in constant
    time using pointer $P_j(x)$ and un-linked from the log;
  Pointer $P_j(x)$ is updated to point to the newly added record.

Note that every log L_{ij} may contain at most one record per data item in the database. Thus, the total number of records in the log vector is bounded by nN, where n is the number of servers and N is the number of data items.

4.3 Auxiliary Data Items and IVVs

When a data item x is copied out-of-bound (i.e., outside the normal update propagation), a separate *auxiliary* copy of this data item, x' is created. A auxiliary copy has its own version vector, which is called auxiliary IVV.

```
SendPropagation(i, V_i):
  if V_i dominates or equals V_j {
    send "you-are-current" message to i and exit;
  }
  for k = 1 to n {
    if (V_jk > V_ik) {
      D_k = Tail of L_jk containing records (x, m) such that m > V_ik;
    else
      D_k = NULL;
    }
  }
  send D and a set S of data items referred to by records in D to i;
end
```

Fig. 2. The *SendPropagation* procedure.

Node i performs all updates to x on its auxiliary copy, while update propagation continues using regular copies. When regular copies "catch up" with auxiliary copies, the latter are discarded.

4.4 The Auxiliary Log

The *auxiliary log*, AUX_i, is used to store updates that i applies to out-of-bound data items. Records in the auxiliary log are of the form $(m, x, v_i(x), op)$, where x is the name of the data item involved, $v_i(x)$ is the IVV that the auxiliary copy of x had at the time the update was applied (*excluding* this update) and *op* is the operation executed (e.g., the byte range of the update and the new value of data in the range). Thus, unlike records in the log vector, auxiliary log records contain information sufficient to re-do the update, and hence they can be much bigger. However, these records are never sent between nodes.

The auxiliary log must be able to support efficiently (in constant time) a function *Earliest(x)* that returns the earliest record in AL_i referring to data item x. Also, we must be able to remove in constant time a record from the middle of the log. These requirements can be easily satisfied with a list structure of the auxiliary log. The details are straightforward and are omitted.

We will often refer to auxiliary data items, their IVVs, and the auxiliary log as auxiliary data structures, as opposed to the rest of the node state, which will be called regular data structures.

5 The Protocol

The protocol consists of procedures executed when a node performs an update, when it propagates updates from another node j, and when it copies a later version of a data item from another node (out-of-bound copying).

5.1 Update Propagation

```
AcceptPropagation(D, S):
  for every x_j in S {
    if v_j(x) dominates v_i(x) {
      adopt x_j from S as a new regular copy x_i;
      v_i(x) := v_j(x);
    else
      declare x_i and x_j inconsistent;
      remove records referring to x from D;
    }
  }
  for each tail D_k from D {
    for every record r from D_k (going from the head of D_k to tail) {
      AddLogRecord(k,r);
    }
  }
end
```

Fig. 3. The *AcceptPropagation* procedure.

Update propagation is done between a recipient node i and a source node j using exclusively regular data structures, regardless of prior out-of-bound copying that might have taken place between the two nodes. For instance, if i had previously copied a newer version of data item x from j as an out-of-bound data and its regular copy of x is still old, x will be copied again during update propagation.[2]

When node i performs update propagation from node j, the following steps are executed:

(1) i sends V_i to j. In response, j executes *SendPropagation* procedure in Figure 2. In this procedure, j compares the received version vector with V_j. If V_i dominates or equals V_j, there is no need for update propagation, and the protocol terminates.

Otherwise, j builds a *tail vector*, D, whose kth component contains log records of updates performed by node k that node i missed, and list S of data items referred to by these records. Note that only regular (non-auxiliary) copies of data items are included into S. j then sends D and S to i. In addition, with every data item x in S, j includes its IVV $v_j(x)$.

(2) When i receives the response from j with D and S, it executes *AcceptPropagation* in Figure 3. It goes through data items from S. For every such data item x_j, i compares x_j's version vector, $v_j(x)$, with the version vector of its local copy of x, $v_i(x)$.

If $v_j(x)$ dominates $v_i(x)$, i adopts the received copy and modifies V_i according to the DBVV maintenance rule 3. Otherwise, i alerts the system administrator that copies x_i and x_j are inconsistent and removes all log records referring to x from all log tails in D (preserving the order of the remaining records). Note that

[2] In other words, out-of-bound copying never reduces the amount of work done during update propagation.

```
IntraNodePropagation:
  for every data item x copied during execution of AcceptPropagation {
    if auxiliary copy of x, x'_i, exists {
      let e = Earliest(x);
      let v_e(x) and op_e be version vector and update operation from e;
      while e ≠ NULL and v_i(x) = v_e(x) {
        apply op_e to x_i;
        v_ii(x) = v_ii + 1;
        V_ii = V_ii + 1;
        append log record (x, V_ii) to L_ii;
        remove e from AUX_i;
        e = Earliest(x)
        let v_e(x) and op_e be version vector and update operation from e;
      }
      if e = NULL {
        if v_i(x) dominates or is equal to v_i(x') {
          remove auxiliary copy x'_i;
        }
      else
        if v_i(x) conflicts with v_e(x) {
          declare that there exist inconsistent replicas of x;
        }
      }
    }
  }
end
```

Fig. 4. The *IntraNodePropagation* procedure.

we do not consider the case when $v_i(x)$ dominates $v_j(x)$ because this cannot happen (see Section 7).

Finally, i appends log tails from the tail vector to the corresponding logs of its log vector, using the *AddLogRecord* procedure from Section 4.2.

Note that removing records that refer to conflicting data items from the tail vector may be an expensive operation. However, this is done only if conflicts are found (which is supposed to be an extraordinary event).

(3) i performs *intra-node update propagation* on Figure 4 to see if any updates accumulated in the auxiliary log can be applied to regular data item.

For every data item x copied in step 2, if auxiliary copy x' exists, i compares the IVV of the regular copy, $v_i(x)$, with the IVV stored in record *Earliest(x)*, the earliest auxiliary log record referring to x.

If both are identical, the operation from this record is applied to the regular copy. All actions normally done when a node performs an update on the regular copy of a data item are executed: $v_{ii}(x)$ and V_{ii} are incremented by 1, and a log record (x, V_{ii}) is appended to L_{ii}. Finally, the auxiliary record *Earliest(x)* is removed from auxiliary log.

If $v_i(x)$ and the IVV of the *Earliest(x)* record conflict, there exist inconsistent copies of x, and conflict is declared.[3]

[3] In fact, the nodes where inconsistent replicas reside can be pinpointed: if the above

The whole process is repeated until either the next earliest auxiliary record has version vector that dominates or conflicts $v_i(x)$, or until the auxiliary log contains no more records referring to x. ($v_i(x)$ can never dominate a version vector of an auxiliary record.) In the latter case, the final comparison of the regular and auxiliary IVVs is done to see whether the regular copy of x has caught up with the auxiliary copy. If so, the auxiliary copy of x can be deleted. (We do not check here whether or not the regular and auxiliary IVVs conflict, deferring conflict detection to the *AcceptPropagation* procedure.)

5.2 Out-of-bound Data Copying

A node i in our protocol can obtain a newer version of an individual data item from any server j at any time. This can be done in addition to regular update propagation of Section 5.1 that causes *all* data at the recipient node to catch up with the data at the source. As already mentioned, data items obtained by direct copying, outside the normal update propagation procedure, are called out-of-bound data items.

Upon receiving an out-of-bound request for data item x, j sends the auxiliary copy x'_j (if it exists), or the regular copy x_j (otherwise), together with the corresponding IVV (auxiliary or regular). Auxiliary copies are preferred not for correctness but as an optimization: the auxiliary copy of a data item (if exists) is never older than the regular copy.

When i receives the response, it compares the received IVV, $v_j(x)$, with its local auxiliary IVV (if auxiliary copy x'_i exists) or regular IVV (otherwise). If $v_j(x)$ dominates, then the received data is indeed newer. Then, i adopts the received data item and IVV as its new auxiliary copy and auxiliary IVV. If $v_j(x)$ is the same as or dominated by the local IVV (auxiliary or regular, as explained above), the received data item is actually older than the local copy; i then takes no action. If the two IVVs conflict, inconsistency between copies of x is declared.

Note that no log records are sent during out-of-bound copying, and the auxiliary log of the recipient is not changed when the old auxiliary copy of x is overwritten by the new data.

5.3 Updating

When a user update to data item x arrives at node i, i performs the operation using auxiliary data structures (if the auxiliary copy of x exists), or regular data structures (otherwise). In the first case, i applies the update to the auxiliary copy x', appends a new record $(x, v_i(x'), update)$ to the auxiliary log, and then modifies the auxiliary IVV: $v_{ii}(x') = v_{ii}(x') + 1$.

In the second case, i applies update to the regular copy x; modifies IVV of the regular copy and DBVV: $v_{ii}(x) = v_{ii}(x) + 1$, $V_{ii} = V_{ii} + 1$; and then appends a log record (x, V_{ii}) to L_{ii}.

---

version vectors conflict in components k and l, then nodes k and l have inconsistent replicas of x.

6 Performance

The procedure *AddLogRecord* is executed in the protocol only when the data item x mentioned in the record being added is accessed anyway. Thus, the location of $P(x)$ is known to *AddLogRecord* for free, and the procedure computes in constant time.

The additional work done by the protocol in the procedure for updating a data item (beyond applying the update itself) takes constant time.

For the rest of this section, we assume that the number of servers is fixed and the size of data items is bounded by a constant. With these assumptions, out-of-bound copying is done in constant time (again, beyond accessing the data items themselves).

Now consider update propagation. In the *SendPropagation* procedure, computing tails D_k is done in time linear in the number of records selected. Since only the records corresponding to updates missed by the recipient are selected, each D_k is computed, at worst, in time linear to the number of data items to be sent (denoted as m). Thus, the total time to compute D is $O(nm)$, where n is the number of servers. Under the assumption that n is fixed, the time to compute D is linear in m.

An interesting question is time to compute set S, which is the union of data items referenced by records in D_k, $1 \leq k \leq n$.

To compute S in $O(m)$, we assume that every data item x has a flag *IsSelected*. The location of this flag is recorded in the control state associated with x. As already mentioned, whenever a log record x is added to the log, the corresponding data item is accessed. Then, the location of x's *IsSelected* flag can be added to the log record at the constant cost.

Then, when *SendPropagation* procedure adds a log record to D_k, the *IsSelected* flag of the corresponding data item x is accessed in constant time. If its value is *"NO"*, the data item is added to S and x's *IsSelected* flag is flipped to *"YES"*. The next time a record referencing x is selected (for a different tail D_l), it will not be added to S. Once computation of D is complete, S will contain the union of data items referenced in records from D_k.

Now, for every data item x in S, its *IsSelected* flag is flipped back to *"NO"*. This takes time linear in the number of data items in S.

Therefore, the total time to compute *SendPropagation* is $O(m)$. In addition, the message sent from the source of propagation to the recipient includes data items being propagated plus constant amount of information per data item. (This information includes the IVV of a data item and the log record of the last update to the data item on every server; recall that regular log records have constant size.)

Finally, the *AcceptPropagation* procedure, in the absence of out-of-bound copying, takes $O(m)$ time to compute (in addition to accessing data items to adopt newer versions received). We conclude that in the common case, the total overhead for update propagation is $O(m)$.

The cost of *IntraNodePropagation* is clearly dominated by the cost of re-applying updates accumulated by the auxiliary copy to the regular copy. This cost is linear in the number of accumulated updates and, depending on the number of such updates, may be high. However, our protocol assumes that few data items are copied out-of-bound. Then, even if overhead imposed by an out-of-bound data item is high, the total overhead is kept low. (Note that *IntraNode-*

Propagation is executed in the background and does not delay user operations or requests for update propagation or out-of-bound copying from other nodes.)

7 Proof of Correctness

Definition 4 Transitive update propagation. Node i is said to perform update propagation transitively from j if it either performs update propagation from j, or it performs update propagation from k after k performed update propagation transitively from j.

Theorem 5. *If update propagation is scheduled in such a way that every node eventually performs update propagation transitively from every other node, then correctness criteria from Section 2.1 are satisfied.*

Proof. See [13].

8 Related work

In this section, we compare our work with existing approaches. Several epidemic protocols have been proposed for replica management. The common feature of existing systems is that they perform anti-entropy and maintain replica consistency at the same data granularity level. Then, as the number of data items grows and the overhead imposed by anti-entropy becomes too large, the existing systems must either schedule anti-entropy less frequently, or increase the granularity of the data (e.g., use a relation instead of a tuple as a granule) to reduce the number of data items.

Neither option is too desirable: the first causes update propagation to be less timely and increases the chance that an update will arrive at an obsolete replica; the second increases the possibility of "false sharing" where replicas are (needlessly) declared inconsistent while the offending updates were actually applied to semantically independent portions of the data item.

Our protocol, on the other hand, decouples the data granularity used for anti-entropy from the granularity used to maintain replica consistency. This enables the system to perform anti-entropy efficiently at the granularity of the entire database, while maintaining replica consistency at the granularity of individual data items.

In the next subsection, we compare our approach with the Lotus Notes system, a commercial epidemic system that is not based on version vectors. The special attention we pay to show how a Lotus Notes-like system would benefit from our approach is partially due to the wide usage of Lotus Notes in practice. We then consider Oracle Symmetric Replication approach in Section 8.2, and replicated database and file systems that employ version vectors in Section 8.3.

8.1 Lotus Notes Protocol

The Lotus Notes protocol [8] associates a *sequence number* with every data item copy, which records the number of updates seen by this copy. Similar to our protocol, Lotus assumes that whole databases are replicated, so that anti-entropy is normally invoked once for all data items in the database. Each server records the time when it propagated updates to every other server (called the last propagation time below).

Consider two nodes, i and j, that replicate a database. Let i invoke an instance of anti-entropy to compare its replica of the database with that of server j, and catch up if necessary. Anti-entropy executes the following algorithm.

1. When node j receives a request for update propagation from i, it first verifies if any data items in its replica of the database have changed since the last update propagation from j to i. If no data item has changed, no further action is needed. Otherwise, j builds a list of data items that have been modified since the last propagation. The entries in the list include data item names and their sequence numbers. j then sends this list to i.
2. i compares every element from the received list with the sequence number of its copy of the same data item. i then copies from j all data items whose sequence number on j is greater.

This algorithm may detect in constant time that update propagation is not required, but only if no data item in the source database has been modified since the last propagation with the recipient. However, in many cases, the source and recipient database replicas will be identical even though the source database has been modified since the last update propagation to the recipient. For instance, after the last propagation between themselves, both nodes may have performed update propagation from other nodes and copied some data modified there. Or, the recipient database may have obtained updates from the source indirectly via intermediate nodes.

In these cases, Lotus incurs high overhead for attempting update propagation between identical database replicas. At the minimum, this overhead includes comparing the modification time of every data item in the source database against the time of the last update propagation. Thus, it grows linearly in the number of data items in the database.

In addition, the first step of the algorithm will result in a list of data items that have been modified or obtained by j since the last propagation. This list will be sent to i, who then will have to perform some work for every entry in this list in step 2. All this work is overhead.

In contrast, the protocol proposed in this paper *never* attempts update propagation between identical replicas of the database. It always recognizes that two database replicas are identical in constant time, by simply comparing their DBVVs.

Moreover, even when update propagation is required, our protocol does not examine every data item in the database to determine which ones must be copied. It makes this determination in time proportional to the number of data items that must actually be copied.

Finally, Lotus update propagation protocol correctly determines which of two copies of a data item is newer only *provided the copies do not conflict*. When a conflict exists, one copy is often declared "newer" incorrectly. For example, if i made two updates to x while j made one conflicting update without obtaining i's copy first, x_i will be declared newer, since its sequence number is greater. It will override x_j in the next execution of update propagation. Thus, Lotus protocol does not satisfy the correctness criteria of Section 2.1.

8.2 Oracle Symmetric Replication Protocol

Oracle's Symmetric Replication protocol [10] is not an epidemic protocol is a strict sense. It does not perform comparison of replica control state to determine obsolete replicas. Instead, it uses a simple approach outlined in the Introduction of this paper. Every server keeps track of the updates it performs and periodically ships them to all other servers. No forwarding of updates is performed.

In the absence of failures, this protocol exhibits good performance. However, a failure of the node that originated updates may leave the system in a state where some nodes have received the updates while others have not. Since no forwarding is performed, this situation may last for a long time, until the server that originated the update is repaired. This situation is dangerous, not only because users can observe different versions of the data at the same time, but also because it increases the opportunity for user updates to be applied to obsolete replicas, thus creating update conflicts.

Our protocol has similar to Oracle performance of update propagation in the absence of failures: it only copies those data items that need to be propagated, executing no work per every data item in the database. However, our protocol does not have the above-mentioned vulnerability to failures during update propagation. If the node that originated updates fails during update propagation (so that some but not all servers received new data), the system will discover this during a periodic comparison of database version vectors on surviving nodes. Then, the newer version of the data items will be forwarded from nodes that obtained it before the failure to the rest of nodes.

The price our protocol pays for this, the periodic comparison of database version vectors, is very small.

8.3 Protocols Using Version Vectors

To our knowledge, version vectors were first introduced in the Locus file system [12] and have been used in several epidemic distributed database and file systems. Version vectors have also been used to prevent out-of-order delivery of causally related messages in a distributed system [3, 6].

The Ficus replicated file system [5] (a descendant of Locus) divides anti-entropy activity into update notification and replica reconciliation processes. Each node i periodically notifies all other nodes about files updated locally. Other nodes then obtain the new copy from i. This notification is attempted only once, and no indirect copying from i via other nodes occurs.

Reconciliation then makes sure that updates have been properly propagated by periodically comparing version vectors of different file replicas and detecting obsolete or conflicting copies. Reconciliation is done on a per data item basis and involves comparing version vectors of every file.

Thus, update propagation in Ficus involves examining the state of every data item, which our protocol avoids. While overall performance of this system is less affected by this than performance of Lotus Notes (since reconciliation may run less frequently because most updates will be propagated by update notification mechanism), our approach would still be beneficial by improving performance of update propagation when it does run.

In Wuu and Berntein's protocol [15], anti-entropy is done by nodes exchanging *gossip messages* [9, 7]. A gossip message from j to i contains log records of updates that j believes are missed by i, and version vector information describing the state of j as well as the extent of j's knowledge about the state of other nodes in the system. The Two-phase Gossip protocol [7] improves [15] by sending fewer version vectors in a gossip message. It also describes a more general method for garbage-collecting log records. Agrawal and Malpani's protocol [1] decouples sending update logs from sending version vector information. Thus, separate policies can be used to schedule both types of exchanges.

The important difference between our protocol and the three protocols above is that the latter perform anti-entropy on the per data item basis, and each invocation involves at least one comparison with the (old) version vector of the recipient copy. Thus, their total overhead is at least linear in the total number of data items.[4]

The protocol proposed in [9] uses version vectors to enforce causally monotonic ordering of user operations on every replica. If an operation arrives out of order, it is delayed until the previous operations arrive. A client stores the version vector returned by last server it contacted and uses it to ensure causal ordering of operations when it connects to different servers.

This approach was extended further in [14]. The protocol of [14] provides more levels of consistency. It also allows to localize consistency control to an individual *session* between a client and the system, independently of other sessions.

These two protocols concentrate on taking advantage of weak-consistency models to improve availability and performance of user operations. Anti-entropy in these systems, as in other existing protocols, is done at the same data granularity as consistency control.[5] Thus, the overhead for anti-entropy in these systems grows, again, linearly with the total number of data items.

9 Conclusion

In this paper, we addressed the scalability issues in epidemic replicated databases. The epidemic approach to replica management is attractive because user operations are serviced by a single (and often a nearby) server, update propagation can be done at a convenient time (i.e., during the next dial-up session), and multiple updates can often be bundled together and propagated in a single transfer.

However, existing epidemic protocols impose overhead that grows linearly with the number of data items in the database. This limits the size of the database that the system can handle without significant performance degradation.

In contrast, the protocol proposed in this paper imposes overhead that is linear in the number of data items that actually must be copied during update propagation. Since this number is usually much lower than the total number of data items in the database, our protocol promises significant reduction of overhead.

[4] In fact, this overhead is even greater because these protocols compare the recipient version vector with every record in the log to be sent. So the overhead is linear in the number of data items plus the number of updates exchanged.

[5] The protocol of [14] uses a database as the granule for consistency control; it does not specify how anti-entropy is done. As already mentioned, doing consistency control at a coarse granularity reduces overhead but increases the possibility of false sharing.

A secondary contribution of this paper is that it explicitly specified correctness criteria for update propagation, separately from correctness criteria for database transactions. This allows one to reason formally about correct propagation regardless of the data consistency guarantees provided by the system to the users.

Finally, we showed how some commercial replicated databases could benefit from our protocol and compared our approach with existing research proposals.

Acknowledgments

The authors thank Garret Swart for reading the paper and verifying its claims about the Oracle replication scheme. We would also like to thank Julie Carroll, Aaron Watters, and Stacey Marcella for their comments.

References

1. D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *The Computer Journal*, 6(34), pp. 534-541, 1991.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.
3. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Comp. Sys.* Vol. 9, No. 3, pp. 272-314, August 1991.
4. 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. In *Proc. of the 6th Symp. on Principles of Distr. Computing*, pp. 1-12, 1987.
5. R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, G. J. Popek, G. J. Rothmeier. Implementation of the Ficus replicated file system. In *Proc. of Usenix Summer Conf.*, pp. 63-71, 1990.
6. C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conf.*, pp. 56-66, 1988.
7. A. Heddaya, M. Hsu, and W. Weihl. Two phase gossip: managing distributed event histories. *Information Sciences*, 49, pp. 35-57, 1989.
8. L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. Presented at the 2d Conf. on Computer-Supported Cooperative Work. September 1988.
9. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Computer Systems,* 4(10), pp. 360-391, November 1992.
10. Oracle 7 Distributed Database Technology and Symmetric Replication. Oracle White Paper, April 1995.
11. D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Software Eng.* 9(3), pp. 240-246, May 1983.
12. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proc. 8th Symp. on Operating Systems Principles*, pp. 169-177, 1981.
13. M. Rabinovich, N. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. AT&T Bell Labs Technical Memorandum 112580-951213-11TM, December 1995.
14. D. Terry, A. Demers, K. Peterson, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems*, 1994.

15. G. T. Wuu and A. J. Bernstein. Efficient solution to the replicated log and dictionary problems. In *Proc. of the 3d ACM Symp. on Principles of Distr. Computing*, pp. 233-242, 1984.