# Optimistic Replication for Internet Data Services

Yasushi Saito and Henry M. Levy
Department of Computer Science and Engineering,
University of Washington, Seattle, WA 98195, U.S.A.
{yasushi,levy}@cs.washington.edu,
http://porcupine.cs.washington.edu

May 21, 2001

## Abstract

We present a new replication algorithm that supports replication of a large number of objects on a diverse set of nodes. The algorithm allows replica sets to be changed dynamically on a per-object basis. It tolerates most types of failures, including multiple node failures, network partitions, and sudden node retirements. These advantages make the algorithm particularly attractive in large cluster-based data services that experience frequent failures and configuration changes. We prove the correctness of the algorithm and show that its performance is near-optimal.

This report originally appeared in the 14th International Conference in Distributed Computing (DISC), October 2000, Toledo, Spain. The proceedings of the conference are published as Lecture Notes in Computer Science No. 1914, Springer Verlag.

# 1 Introduction

We present a new lightweight replication algorithm designed for PC-based Internet data services, such as WWW, FTP, and email. Our algorithm is unique in many aspects. First, our algorithm lets copies of an object, or *replicas*, be created or deleted dynamically and yet guarantees that the state of all the replicas eventually converges. Second, it is designed specifically to support many small replicated objects, which is typical in web-based environments; in particular, it has low space and computation overhead and handles object deletion efficiently. Third, the algorithm tolerates most failure types, including multiple node failures and sudden node retirements, and network partitions. Finally, it is non-blocking and decentralized, that is, it lets any node issue an update, and it makes no single node permanently responsible for maintaining replica consistency. We achieve this efficiency and versatility by being *optimistic*, that is, we guarantee that replicas become consistent only eventually. This optimism precludes the use of our scheme in applications that demand high data reliability, such as banking, but it poses little problem in typical Internet services, because these services have simple update semantics and weak consistency requirements.

## 1.1 Background

Our work derives from the Porcupine cluster-based mail server project [19]. Porcupine connects up to several hundred off-the-shelf computers to serve billions of mail messages per day. Porcupine's architecture is fully *dynamic*; any node can potentially manage any user's profile and store any user's email messages. For each incoming message, Porcupine chooses a set of nodes on which to replicate and store the message (called a *replica set*), based on node load and message affinity. A cluster membership service and a distributed naming service keep track of the locations of users' messages. The dynamic message placement yields many benefits: performance improvement via balanced load and flexible support for system configuration changes by message migration.

From the viewpoint of a replication service, Porcupine presents an environment very different from traditional database systems. Following are the key characteristics of Porcupine and a discussion of how they define the goals of our replication service:

**Frequent failures:** With hundreds of nodes in the cluster, a part of the system is always down. First, the algorithm must provide ***strong fault tolerance*** by maintaining replica consistency even when some of the nodes are down, sometimes permanently. Second, the algorithm must be ***non-blocking***, that is, it must allow reads and writes to any replica any time regardless of whether peer replicas are reachable.

**Changing replica sets:** Porcupine needs to change email message replica sets automatically to react to node additions and removals. We must support ***dynamic addition***

***and removal of replicas*** while allowing contents updates to the object.

**Small object size:** The unit of replication in Porcupine is an email message whose average size is 5K bytes, as opposed to many gigabytes typical in database systems. We need to ***minimize the space overhead*** of per-object data structures used to maintain replica consistency.

**Selective replication:** Porcupine stores billions of email messages, each of which is in its own replica set. Our algorithm needs to be ***quiescent***, that is, it should incur no computational and space overhead when no update is in progress. Moreover, email messages are deleted frequently. Thus, our system should support ***quick object deletion*** without leaving any data structures behind.

**Weak consistency requirements:** Services such as email do not demand strict replica consistency because the possibility of inconsistent data is inherent in the environment; for example, unreliable network transport can cause delivery delay or duplicate messages. Thus, we only need to support ***eventual consistency*** of replica state.

These service requirements are not unique to email — in fact, they are shared by many other Internet applications, including Usenet [20], Internet-based BBS services (e.g., slashdot.org or delphi.com), naming services [15, 13], and wide-area mirroring of Web or FTP data [17]. All these services are potential applications of our replication algorithm.

## 1.2 Related Work

Data replication has been studied and deployed widely. However, previous work in this area has addressed the goals of our algorithm only in a piecemeal fashion and has not solved all of the problems that we face in our intended environment.

Traditional replication algorithms include primary-copy algorithms used widely in commercial database systems [3], quorum consensus algorithms [7, 9, 10], and atomic broadcast protocols [4, 1]. They try to achieve single-copy semantics, that is, they give users an illusion of having a single, highly available copy of an object. Although generic, these algorithms fail to address problems we face – frequent failures and frequent changes – because they sacrifice availability by prohibiting accesses to a replica when data is not provably up to date.

Mobile replicated database systems (e.g., Bayou [16], and Roam [18]) share some of our goals: elimination of a single point of failure, handling frequent failures, and dynamic replica addition and deletion. The main difference between their solutions and ours is that these systems focus on minimizing the communication overhead, whereas we focus on minimizing the space and the computation overhead. For example, the techniques used by these systems, including on-demand polling and a semantic log for describing updates, reduce the communication cost but increase both the computation and the storage overhead.

Our algorithm is most closely related to multi-master wide-area replicated services, including Active Directory [13], Xerox Clearinghouse [6], and Usenet [20]. These systems provide non-blocking accesses, support replication of many small objects, and propagate updates efficiently over unreliable links. However, they do not support replica set changes and provide only a weak fault tolerance; for example, one failed node can stall the update propagation of the entire system. Moreover, the existing systems do not support quick object deletion and require storing update records (often called "death certificates") for an indefinitely long period.

Several systems allow dynamically changing the placement of replicas using reference-monitoring mechanisms to balance the system load [17, 23]. They update replicas by gossiping changes along a spanning tree and are unable to achieve replica consistency even under a single node failure. Our work complements them by proposing a robust mechanism that can tolerate a wider variety of failures.

## 1.3 Overview of the Algorithm

Our algorithm is based on three principles: *state transfer*, update resolution using *Thomas write rule*, and update retirement using *synchronized clocks*.

In its basic form, our algorithm is similar to systems such as Active Directory [13] and Usenet [20]. Any replica (or any node for a newly created object) can issue an update any time. A coordinator, usually the issuer of the update, propagates the update by pushing the new object state to others in background. Conflicting updates are resolved by Thomas write rule [21], that is, by attaching timestamps to them and accepting only the newest update.

Our algorithm is unique in its uniform handling of replica set updates and contents updates — in fact, an update is actually a tuple consisting of the new object contents and the new replica set. For an update that changes the replica set, the coordinator pushes the update to the union of the old and the new replica sets (called the *targets* of the update). A node receiving the update either modifies, creates, or deletes a replica depending on whether or not it appears in the new replica set. Thomas write rule is again used to resolve conflicts among replica set changes; the older updates are canceled by forwarding the newest update to their targets and letting them be rolled back. Overriding the older updates requires the coordinator of the newer update to discover the older updates' targets. This *node discovery* problem is similar to the distributed resource discovery problem [8], and the solution is also similar: the

nodes that receive the update send back to the coordinator the sets of nodes they know and let the coordinator expand its target node set transitively.

We apply at-most-once messaging technique using synchronized clocks [12] to retire updates. After the coordinator completes update propagation, it sends out retirement notices to the target nodes. Upon receiving a retirement notice, a node deletes the update from disk after waiting for a fixed period; the wait ensures that the node will never apply a stale update in the future.

This combination of state transfer, Thomas write rule, and quick update retirement allows our algorithm to solve our goals effectively as follows:

- Our algorithm's basic design directly achieves the three goals — non-blocking access, dynamic replica set changes, and eventual consistency.
- We achieve strong fault tolerance in two ways. First, our algorithm is fully decentralized — in particular, it lets any node take over the task of the coordinator any time. Second, its node discovery process eliminates a single point of failure quickly and lets the system tolerate a sudden node retirement without compromising replica consistency.
- Our algorithm's space overhead is quite small for two reasons. First, the state transfer architecture minimizes the size of the update record by omitting the new object contents – the object contents are usually read from the replica directly. Second, our algorithm quickly reclaims the space occupied by update records by retiring them as soon as they finish propagation.

### 1.4 Structure of the Paper

The rest of the paper is structured as follows. We describe our algorithm in detail in Section 2 and show two examples in Section 3 to elucidate its behavior, in particular, the resolution of concurrent updates. Section 4 proves the correctness of our algorithm. In Section 5, we discuss several extensions to the basic algorithm to address issues that arise in practice: e.g., optimizations to make the algorithm work efficiently and the handling of long-term failures. We briefly discuss the computational and the space overhead of the algorithm in Section 6 and conclude in Section 7.

## 2 The Replication Algorithm

Although our algorithm is designed to support many objects replicated on a diverse set of nodes, it is first presented in the context of a single object. We describe a straightforward extension of the basic algorithm to support multiple objects in Section 5.1.

```
type Timestamp = record
  time: Clock    // wall-clock time.
  nid: NodeID  // a tie-breaker.
end
type Update = record
  state: {ACTIVE, RETIRING, RETIRED, SUSPENDED}
  ts: Timestamp
  target, done, peer: Set⟨NodeID⟩
end
```

**Figure 1**. *Data structures used by the algorithm.*

### 2.1 System Model and Assumptions

We make the following assumptions about the environment. First, the nodes in the system can communicate through a fully connected network. Second, nodes and network links may crash, and messages may be reordered, delayed, or lost, but Byzantine failures will not occur. Finally, the nodes have loosely synchronized clocks; clock synchronization algorithms are well known and are deployed widely [14].

Our algorithm only propagates updates. Locating and reading replicas and choosing the replica placement are outside the scope of this algorithm. For locating replicas, any weakly consistent naming service can be used. For replication policy, an object could be assigned to a random set of nodes [19] or to a set determined by reference-monitoring mechanisms [17, 23].

### 2.2 Notational Conventions

All global variables are stored on stable storage and survive node crashes. Procedures marked **public** run as transactions that are non-preemptive and update global variables atomically. $\langle val_1, val_2 \rangle$ is a tuple of two values. Send(*node, proc, args...*) sends a message to *node* and requests calling the procedure *proc* with *args...*. Send does not wait for *proc* to finish; it merely queues the message. Texts after ' // ' are comments.

### 2.3 Data Structures

Fig. 1 shows the types used in the algorithm. Loosely synchronized clocks order updates [14]. Other types of clocks, e.g., logical clocks [11], may be used without affecting the correctness of the algorithm, but wall clocks best suit our purpose because they can order logically unrelated events (e.g., a user contacting two nodes in a cluster serially). The procedure Now() returns the current local clock value. We assume the clock resolution to be fine enough that successive calls to

```
var
  gData: Content←NULL
  gPeer: Set⟨NodeID⟩←φ
  gU: Update←NULL
  gRetireTime: Clock
  gSavedData: Content
```

**Figure 2**. *Per-node, per-object global variables used by the algorithm.*

```
public proc UpdateObject(peer, data)
  u←Update(ts← Timestamp(Now(), me), state←ACTIVE,
           done←φ, peer←peer, target←peer)
  ApplyUpdate(u, data)
```

**Figure 3**. UpdateObject *is called by the application to create a new object, modify the object contents, add or remove the replica set, or delete the object.*

Now() always return different values; this assumption lets us use timestamps to identify updates.

An update to the object is represented by the **Update** record. Its *state* field indicates the update's status. A new update starts as being ACTIVE. An update is RETIRING on the coordinator while retirement notifications are sent, and it is RETIRED after the reception of a retirement notice. An update is SUSPENDED when it is found to conflict with a newer update, and it stays dormant until the newer update arrives and supersedes it. *Target*, *done*, and *peer* fields specify the set of nodes that should receive the update, have acknowledged the update, and should replicate the object, respectively. Thus, *done* and *peer* are always subsets of *target*. The update propagation finishes when *done* = *target*.

Five persistent variables are stored per replica on a node (Fig. 2). Two of them, *gData* and *gPeers*, are visible to the application and the rest are used internally by the replication algorithm. *GData* stores the actual contents of the object — we are not concerned about the object's internal structure in this paper. *GPeer* shows, to the best of the node's knowledge, the replica set of the object. *GU* remembers the newest update applied on the object. Notice the absence of the new object contents in *gU* — the contents are propagated to other nodes by reading from *gData* directly most of the time. The exception is when *gData* is deleted by an update, but the object contents still need to be propagated to other nodes (this happens, for example, when the object is moved from one node to another). *GSavedData* is used to save the new object contents in such cases, and it is otherwise null. That *gSavedData* is usually null contributes to reducing the space overhead of the algorithm, because all other data structures used by the algorithm are of small and fixed size. *GRetireTime* is used to delete a retired update and is discussed further in Section 2.7.

## 2.4 Application Programming Interface

One procedure, UpdateObject (Fig. 3), is called by the application. It takes two parameters, the new replica set (*peer*) and the new object contents (*data*). Passing an empty set to *peer* will delete the object entirely from the system. The caller of this procedure must ensure that the node stores a replica already, except when the object is being newly created. This restriction, also discussed in Section 4.2, is to prevent creating an orphan replica that is disconnected from others and is not found by the node discovery process. The implicit variable *me* shows the name of the node itself.

## 2.5 Update Application

The procedure ApplyUpdate (Fig. 4), called both from the local application and from remote nodes, logs and applies the update to the replica and prepares for update propagation. It first merges the target sets of both *u* and the current update, *gU* ⟨1⟩[4]. This must be done even when *u* is to be discarded ⟨2⟩ so that the participants of both updates can eventually receive the newer update.

## 2.6 Update Propagation

An update is pushed to other nodes periodically by PushApply (Fig. 5). The target node set expands as replies come back from the target nodes ⟨6⟩. The propagation finishes when all the target nodes reply ⟨4⟩.

The function IAmCoordinator tells whether the node is designated to coordinate a particular update. For now, it just returns true, meaning that any node can be a coordinator, and that an update is flooded among all the target nodes. Having multiple coordinators does not affect the correctness of the algorithm, but it surely wastes the network bandwidth — we improve this design in Section 5.2.

## 2.7 Deleting Retired Updates

While each update record occupies only a small space, it is stored even when the replica itself is removed. We need to delete updates in a timely manner; otherwise, the update records of deleted replicas will accumulate and eventually fill the disk up. An update is deleted from disk in two steps. The first step, performed periodically by PushRetire (Fig. 6), is the update *retirement*; the coordinator informs the target nodes that update propagation is complete. The second step is the update *removal* in which we apply the at-most-once messaging algorithm [12] to remove retired updates without being confused

[4] Markers such as ⟨1⟩ refer to lines in the program listings.

```
proc ApplyUpdate(u, data): bool
  // Expand knowledge. ⟨1⟩
  u.target ← u.target ∪ gPeer
  if gU ≠ NULL ∧ gU.state ∉ {RETIRING,RETIRED}
    u.target ← u.target ∪ gU.target
    gU.target ← u.target
  // Reject if u is stale. ⟨2⟩
  if gU ≠ NULL ∧ gU.ts > u.ts
    return false
  // Log the update. ⟨3⟩
  gU ← u
  gSavedData ← NULL
  // Modify the replica.
  if me ∈ gU.peer
    ⟨gData, gPeer⟩ ← ⟨data, gU.peer⟩
  else
    ⟨gData, gPeer⟩ ← ⟨NULL, ϕ⟩
    if gU.peer ≠ ϕ
      // Save data; not needed when peer is
      // ϕ since everyone deletes the replica.
      gSavedData ← data
  gU.done ← gU.done ∪ {me}
  return true
```

**Figure 4**. *Local update application. This procedure logs and applies the update to the local replica. It returns whether the update was successfully applied or not.*

by out-of-order update messages. Here, the node simply waits for *WAIT* seconds before deleting a retired update (Fig. 6, RemoveUpdate). *WAIT* is the sum of the maximum clock skew among nodes and the message lifetime, an interval long enough that almost all the messages will arrive at the destination within the interval.

This update removal scheme additionally requires each node to discard stale incoming network messages (Fig. 6 MessageArrived). Here, each network message is stamped with the sender's clock value and is accepted by the receiver only when its timestamp is no older than *WAIT* on the receiver's clock.

# 3 Examples

We show two examples to illustrate the behavior of the algorithm. The first example is a simple contents update. The second example demonstrates the node discovery process used to resolve conflicting replica set changes.

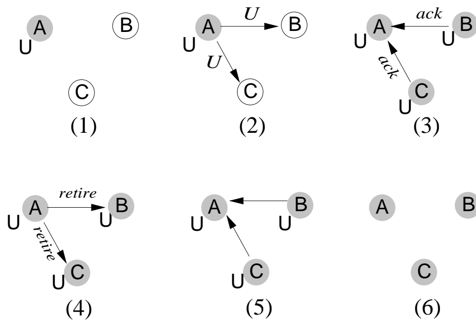


**Figure 7**. *An object replicated on nodes A, B, and C is updated. Gray circles indicate that nodes that have applied the update. The letter "U" indicates that the update is logged on the node.*

Fig. 7 shows a sequence of steps performed to update an object replicated on nodes A, B, and C.

(1) A issues an update $U$:⟨target=peer={A,B,C}⟩ and modifies its replica.

(2) A pushes $U$ to B and C.

(3) B applies $U$ and returns ⟨*U.ts*, true, {A,B}, {A,B,C}⟩ to A. C applies $U$ and returns ⟨*U.ts*, true, {A,C}, {A,B,C}⟩ to A.

(4) A receives the replies from B and C and changes *U.state* to RETIRING. A sends $U$'s retirement to B and C.

(5) B and C change *U.state* to RETIRED and reply to A. A receives the replies from B and C and changes *U.state* to RETIRED.

(6) *WAIT* seconds later, A, B, and C erase $U$ from *gU*.

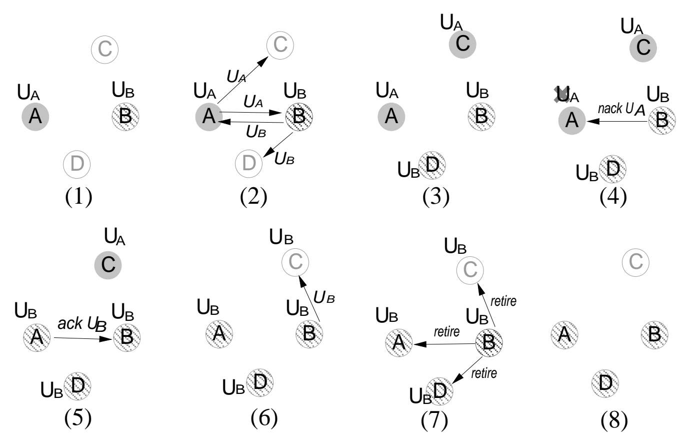


**Figure 8**. *Conflicting updates involving replica addition. Gray circles are nodes that applied $U^A$, and diagonally shaded circles are nodes that applied $U^B$.*

Fig. 8 shows a scenario in which an object replicated on nodes A and B is updated concurrently, first by A that adds C to the replica set, and next by B that adds D to the replica set. We assume that B's update is newer than A's.

(1) A issues $U^A$:⟨target=peer={A,B,C}⟩ and modifies its *gPeer*. Simultaneously, B issues $U^B$: ⟨target=peer={A,B,D}⟩ and modifies its *gPeer*.

(2) A pushes $U^A$ to B and C. B pushes $U^B$ to A and D. Now, for the sake of explanation, suppose C and D receive the updates before B and A do.

(3) C creates a replica and replies ⟨$U^A$.*ts*, true, {A,C}, {A,B,C}⟩ to A. D creates a replica and replies ⟨$U^B$.*ts*, true, {B,D}, {A,B,D}⟩ to B.

(4) B receives $U^A$ from A. Because $U^A.ts < U^B.ts$, B rejects $U^A$ and replies ⟨$U^A$.*ts*, false, {A,B} {A,B,C,D}⟩ to A. B's $U^B$.*target* becomes {A,B,C,D} (Fig. 4 ⟨1⟩). On receiving the reply from B, A changes $U^A$.*state* to SUSPENDED.

(5) A receives $U^B$ from B. Because $U^B.ts > U^A.ts$, A modifies *gPeer*, replaces *gU* with $U^B$, and replies ⟨$U^B$.*ts*, true, {A,B}, {A,B,C,D}⟩ to B. Later, A may receive a reply for $U^A$ from C, but A ignores the reply (Fig. 5 ⟨5⟩).

(6) B receives the reply from A and pushes $U^B$ to C, the only target not yet contacted. On reception of $U^B$, C recognizes that it is not a part of the new replica set, removes its replica, and replies ⟨$U^B$, true, {A,B,C}, {A,B,C,D}⟩ to B.

(7) B receives the replies from C and D. B changes $U^B$.*state* to RETIRING and pushes $U^B$'s retirement to A, C, and D. A, C, and D acknowledge the retirement.

(8) *WAIT* seconds later, A, B, C, and D erase $U^B$ from *gU*. In the end, A, B, and D store replicas. C created a replica and later deleted it.

# 4 Correctness Proof

While being simple, our algorithm contains several subtleties, especially regarding replica additions and deletions. For example, how does it guarantee that all replicas receive an update, when another update is adding replicas concurrently? In this section, we prove two main safety properties of the algorithm: all nodes receive the newest update at the end of propagation, and no node accepts stale updates regardless of concurrent updates. We also argue the liveness of the algorithm, i.e., all the replicas will receive the newest update, by applying our safety arguments.

The state of the system can be viewed as a directed graph, called the *knowledge graph*, in which the vertices represent nodes and the edges represent the nodes' knowledge of others through *gPeer* and *gU.targets*. The graph is usually complete (i.e., no update is issued and the value of *gPeer* is identical on all the replicas), but it becomes incomplete during replica addition or deletion. At a high level, our proof shows that the algorithm ripples the newest update through the graph, adding edges to the graph along the way to cover all the nodes and to restore the completeness of the graph eventually.

Following are notations used in the ensuing proof. Other symbols are summarized in Table 1.

- A node "*stores a replica*" when *gData*≠NULL.
- A node "*has an update*" when its *gU.state* is ACTIVE or RETIRING.
- A node "*retires an update*" when it sets *gU.state* to RETIRED.
- A node $n_1$ "*knows*" another node $n_2$ when either $n_1$ has an update and $n_2 \in n_1$:*gU.target*, or $n_2 \in n_1$:*gPeer*.
- $G_T \equiv \langle G_T, E_{G_T} \rangle$, a *knowledge graph* for the object at time $T$ [5], is defined as follows.

  $G_T =$ Nodes that store a replica or have an update.
  $E_{G_T} = \{v_1 \rightarrow v_2 \mid \{v_1, v_2\} \subseteq G_T \wedge v_1 \text{ knows } v_2\}$
- $S_T \equiv \langle S_T, E_{S_T} \rangle$, an *induced subgraph* of $G_T$, excludes from $G_T$ vertices that correspond to failed nodes and the associated edges. $S_T$ shows the knowledge graph in the presence of failure.

## 4.1 Correctness Criteria

Ideally, we want to prove that the algorithm keeps all the live replicas consistent regardless of types of failures. Such a guarantee, however, is impossible when nodes or links fail in a way that makes the corresponding induced subgraph disconnected. For example, suppose two nodes fail simultaneously after both have created two new replicas, as illustrated in Fig. 9. After such a failure, any update issued on the two new replicas will not reach each other, and the new replicas will remain inconsistent until the original two nodes recover. Therefore, we define the correctness only under the condition that a knowledge subgraph is at least weakly connected.

> **Correctness criteria:** *Suppose $S_{T_s}$ is weakly connected, and no node or link fails and no new update is issued during a long enough period $(T_s, T_e)$. Let $U$ be the newest update generated before $T_e$. The algorithm is correct if the following conditions hold.*
>
> *(1) Every node $n \in U$.peer applies $U$ before $T_e$.*
>
> *(2) No node $n \notin U$.peer stores a replica at $T_e$.*

[5] All times mentioned in the proof are hypothetical global times observed by an external agent.

| Symbol | Meaning |
|---|---|
| $n{:}var$ | The value of variable $var$ on node $n$. |
| $n{:}U.target_{,T}$ | The value of $U.target$ on node $n$ just before $T$. |
| $n{:}U.done_{,T}$ | The value of $U.done$ on node $n$ just before $T$. |
| $n_1 \stackrel{G_T}{\rightarrow} n_2$ | $n_1$ knows $n_2$ in $G_T$, i.e., $(n_1 \rightarrow n_2) \in E_{G_T}$. |
| $n_1 \stackrel{G_T}{\rightsquigarrow} n_2$ | A path from $n_1$ to $n_2$ exists, i.e., $n_1 \stackrel{G_T}{\rightarrow} n_2 \vee \exists n', n_1 \stackrel{G_T}{\rightarrow} n' \wedge n' \stackrel{G_T}{\rightsquigarrow} n_2$. |

Table 1. Notational conventions used in the proof

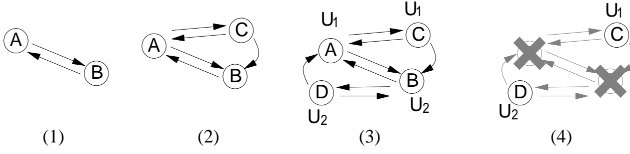


**Figure 9**. *A scenario that disconnects a graph. The edges show knowledge among nodes. (1) Replicas A and B initially know each other. (2) A issues $U_1$ and creates a replica C. (3) B issues $U_2$ and creates a replica D. (4) A crashes before $U_1$ is propagated to B or D. B crashes before $U_2$ is propagated to A or C. In the end, two components, $\{C\}$ and $\{D\}$, are both live but disconnected.*

(3) *No update $U'$ older than $U$ (i.e., $U'.ts < U.ts$) is applied on any node after $U$ is applied.*

Notice that these criteria demand, in case of a graph disconnection, a replica consistency within each partition, and that as soon as the partitions re-integrate, all the replicas converge onto the globally newest state. Indeed, this set of criteria is as strong as any non-blocking replication algorithm can guarantee.

## 4.2 Graph Invariants

**Theorem 1** *If $S_T$ is strongly connected, then $\forall T' > T$, $S_{T'}$ is also strongly connected if no node or link fails during the period $(T, T')$.*

**Theorem 2** *If $S_T$ is weakly connected, then $\forall T' > T$, $S_{T'}$ is weakly connected if no node or link fails during the period $(T, T')$.*

**Proof sketch.** We show by induction that no transition on the subgraph can disconnect the subgraph. First, a replica creation will not disconnect the graph because a new replica is always created by "stemming out" from an existing replica (Section 2.4). Next, replica deletion does not change the graph shape because the update record is still stored on the same node. Finally, update retirement will not disconnect the graph because an update retires only after all the peer replicas have spanned the edges to one another. Theorem 2 can be proved exactly the same way. ■

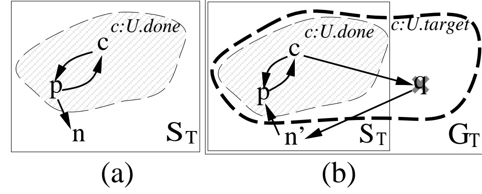


**Figure 10**. *A coordinator c may fail to contact a node n in two situations.*

**Theorem 3** *$G_T$ is strongly connected for all $T$.*

**Proof.** The graph is clearly connected in the base case in which no replica exists. Thus, $G_T$ is connected for all $T$ from Theorem 1. ■

## 4.3 All Replicas Receive the Newest Update

Now, we prove that all the nodes receive the newest update by distinguishing two cases. Theorem 4 proves that if an update retires, all the nodes must have received the update. Theorem 5 proves that when an update is unable to retire because some of its targets are dead, all the remaining nodes still receive the update. These two theorems together prove the correctness criteria (1) and (2).

**Theorem 4** *Suppose $S_{T_s}$ is weakly connected, and no node or link fails and no new update is issued during a long enough period $(T_s, T_e)$. Let $U$ be the newest update generated before $T_e$. If a coordinator $c$ begins the retirement of $U$ at time $T$ $(T < T_e)$, then $S_{T_e} \subseteq c{:}U.done_{,T}$; that is, all the nodes in $S_{T_e}$ have received and applied $U$.*

**Proof.** We only need to prove that $S_T \subseteq c{:}U.done_{,T}$; if $S_T \subseteq c{:}U.done_{,T}$, then $S_{T_e} \subseteq S_T$ because no newer update is generated after $T$ and no node possesses a stale update after $T$.

For the sake of contradiction, suppose $S_T \not\subseteq c{:}U.done_{,T}$.

Because $S_T$ is weakly connected from Theorem 2, we can pick a node $p \in c{:}U.done_{,T}$ such that $\exists n \in S_T - c{:}U.done_{,T}$ and that either $n \stackrel{S_T}{\rightarrow} p$ or $p \stackrel{S_T}{\rightarrow} n$. Below, we show that no such pair of $p$ and $n$ can exist.

First, suppose a pair $(p, n)$ with an edge $p \stackrel{S_T}{\rightarrow} n$ exists (Fig. 10 (a)). Let $T_p$ be the time $c$ propagated $U$ to $p$ $(T_p < T)$. First,

the edge $p \xrightarrow{S_T} n$ must have been created at or before $T_p$, because $U$ is the newest update and any other update that could have created $p \rightarrow n$ would have been rejected by $p$ after $T_p$. On the other hand, $p \xrightarrow{S_T} n$ must have been created after $T_p$; otherwise, the edge $c \xrightarrow{S_T} n$ must be in the graph (Fig. 5 ⟨6⟩). Because of this contradiction, this pair $(p,n)$ cannot exist. Second, suppose only a pair $(p,n')$ with an edge $n' \xrightarrow{S_T} p$ exists (Fig. 10 (b)). From Theorem 3, a path $c \overset{G_T}{\rightsquigarrow} n'$ exists in the full graph $G_T$ (remember, $G_T$ may include dead nodes). Therefore, there exists a dead or uncommunicative node $q \in c{:}U\mathit{target}_{,T}$ along the path $c \overset{G_T}{\rightsquigarrow} n'$, and $q$ makes $U$ unable to retire in the first place. Therefore, this pair nodes $(p,n')$ cannot exist as well. Therefore, for $U$ to retire, $S_T \subseteq c{:}U\mathit{done}_{,T}$. ■

**Theorem 5** *Suppose $S_{T_s}$ is weakly connected, and no node or link fails and no new update is issued during a long enough period $(T_s, T_e)$. Let $U$ be the newest update generated before $T_e$. If $(c{:}U\mathit{target}_{,T_e} - c{:}U\mathit{done}_{,T_e}) \cap S_{T_e} = \phi$ on a coordinator node $c$, then $S_{T_e} \subseteq c{:}U\mathit{done}_{,T_e}$.*

**Proof.** For the sake of contradiction, suppose $S_{T_e} \not\subseteq c{:}U\mathit{done}_{,T_e}$. Using the argument that appeared in the previous proof, we can pick a node $p \in c{:}U\mathit{done}_{,T_e}$ such that $\exists n \in S_{T_e} - c{:}U\mathit{done}_{,T}$ and either $n \xrightarrow{S_{T_e}} p$ or $p \xrightarrow{S_{T_e}} n$, and we can show that a pair $(p,n)$ with an edge $p \xrightarrow{S_T} n$ cannot exist.

Now, suppose only a pair $(p,n')$ with an edge $n' \xrightarrow{S_T} p$ exists. For the moment, lets assume the following lemma holds for any pair of nodes, $n_1$ and $n_2$.

> **Lemma 1** *If $n_1 \xrightarrow{S_T} n_2$ but not $n_2 \xrightarrow{S_T} n_1$, then $n_1$ has an update.*

From this lemma, $n'$ has an update, say, $U'$. Thus, $n'$ contacts $p$, which in turn causes $n'$ to discover $c$ (Fig. 5 ⟨6⟩), which in turn cause $n'$ to propagate $U'$ to $c$, thereby letting $c$ discover $n'$ before $T_e$ (Fig. 4 ⟨1⟩). Therefore, a pair $(p,n)$ with an edge $n \xrightarrow{S_T} p$ cannot exist as well. Thus, $S_{T_e} \subseteq c{:}U\mathit{done}_{,T_e}$. ■

We sketch the proof of Lemma 1. Edges can disappear only when an update retires. For an update to retire, all the target nodes need to reply, that is, edges must span between any pair of the target nodes. Thus, when $n_1 \xrightarrow{S_T} n_2$ but not $n_2 \xrightarrow{S_T} n_1$, then $n_1$ must have an update. ■

### 4.4 No Node Receives a Stale Update

**Theorem 6** *Suppose no update is generated for a long period ending at $T_e$. Let $U$ be the newest update issued before $T_e$. After $U$ retires, no update older than $U$ is applied on any node.*

```
var members: Set⟨NodeID⟩;
proc IAmCoordinator(u): bool
  return u.ts.nid = me
       ∨ min(members ∩ u.done) = me
```

**Figure 11**. *Designated coordinator selection. A membership service stores the set of presumed live nodes in members.*

**Proof.** A node may receive an older update after it retires $U$ for two potential reasons: (1) another node that has not received $U$ propagates a stale update, or (2) a network delay causes a message containing a stale update to be received after $U$ retires. Theorem 4 prevents the case (1). The use of synchronized clocks, described in Section 2.7, prevents the case (2) (refer to [12] for the full proof).

### 4.5 Liveness

Liveness is derived immediately from the work of the algorithm. The algorithm sends update or retire messages to the nodes in *gU.target* until it receives the replies from all the nodes. *GU.target* may grow as a result of reply processing (Fig. 4 ⟨1⟩), but because its size is finite, the coordinator will eventually push the update to all the nodes it can communicate with[6].

## 5 Extensions

### 5.1 Supporting Multiple Objects

All the discussions so far have focused on a single object, but in fact, the basic algorithm can be extended easily to support multiple objects. To support multiple objects, instead of the variables *gU*, *gSavedData*, and *gRetireTime*, we now have a persistent table that partially maps an object ID to an update in progress for the object. An update is added to the table when an object is going to be modified (Fig. 4 ⟨3⟩), and is deleted from the table when it is removed (Fig. 6 ⟨7⟩) or is superseded by a newer update for the same object (Fig. 4 ⟨3⟩).

### 5.2 Designated Coordinator

The basic algorithm presented so far is inefficient because it floods an update among all the target nodes in PushApply, causing an update to be sent as many as $(N-1)^2$ times ($N$ is the number of replicas). By combining the use of a group membership service [5, 22] and a simple change to the function IAmCoordinator (Figures 5 and 11), however, we can reduce

[6]Here, we assume a bounded message transmission delay. Otherwise, no algorithm can ensure liveness.

the cost to $N-1$ in the common case. In the new implementation, a node pushes or retires an update only when it is the issuer of the update or when it is designated to take over the failed issuer. Notice that because the membership service is shared by all the objects hosted on a node, its cost is amortized over many runs of the algorithm and becomes negligible.

### 5.3 Delaying Update Retirements

The algorithm is further optimized by delaying and aggregating calls to PushRetire for different objects to the same node. Delaying calls to PushRetire does not affect the replica consistency; it merely delays the deletion of the update record and increases the size of the update table.

### 5.4 Optimistic Deltas

Instead of pushing the entire object state every time, we can send *optimistic deltas* [2] to save the network and the computational cost. Here, a coordinator simply pretends that all the replicas for the object were consistent before the update and pushes only the difference between the old and the new contents (called the optimistic delta) along with the *fingerprint* of the old replica contents. On the receiver side, a node applies the update when its replica's fingerprint matches the update's; otherwise, the node requests a full contents transfer from the coordinator. This technique can reduce the cost of the algorithm, especially during replica set changes, in the common case without concurrent updates.

Fingerprint is any short bit-string that summarizes the replica contents. Applying a collision-resistant hash function (e.g., MD5) on the replica contents is one way to compute a fingerprint. A faster, more accurate, but slightly more space-consuming alternative is to store along with each replica a timestamp (Figures 1 and 4) that shows the last time the replica was modified, and to use the timestamp as a fingerprint.

### 5.5 Handling Long-term Failures

In the real world, computers often crash and never recover. Such nodes create an unbounded amount of backlog of updates that eventually fill up the disks on other nodes. Our algorithm handles such a situation automatically by purging nodes that remain down for too long.

When a node finds another node dead for more than a predefined *purge period* (e.g., one week), it pretends that it received affirmative replies from the dead node for all its jammed updates. The node then purges the dead node simply by removing the dead node's name from the replica sets of all the replicas stored on the node. To avoid having inconsistent data, when a node recovers after being down for the purge period or longer, it clears its disk contents and rejoins the cluster with a new node ID.

The only remaining problem is when nodes or links fail in such a way as to make the knowledge graph disconnected, and they remain failed until the purge deadline. In such case, the aforementioned scheme may make replicas permanently inconsistent. We argue below that such a scenario is highly unlikely to happen in practice.

How can a graph become disconnected? One cause of a graph disconnection is link failures (i.e., network partitioning). Another cause, which may happen without link failures, is multiple node failures combined with concurrent replica additions, illustrated in Fig. 9. Now, can a graph disconnection last until the purge period? The answer is no, for all practical purposes. First, a network partitioning would never last long because it is repaired simply by installing replacement parts. Second, the latter failure scenario would not happen in practice because it requires a combination of simultaneous replica creations and coincidental sudden long-term failures of multiple nodes — the window of vulnerability is very narrow for both.

# 6 Performance

### 6.1 Networking and Computational Overhead

With the optimizations described in Section 5.2, our algorithm pushes an update to $N$ replicas in the common case and aggregates retirement notices into one batch notice. In total, the algorithm sends $2(1+\frac{1}{G})N$ messages per update, where $G$ is the average aggregation factor for retirement notices. In Porcupine, the value of $G$ is around 20 under heavy load, and the networking and the processing costs of our algorithm is close to $2N$ per update, which is the optimal number for an algorithm that does not batch (and thus delay) update propagation — $N+N$ messages are always needed to propagate and acknowledge an update to $N$ nodes.

### 6.2 Space Overhead

This algorithm stores two types of data structures per replica in addition to the contents: the replica set (*gPeer* in Fig. 1), and the update record (*gU*, *gSavedData*, and *gRetireTime* in Fig. 1).

The replica set information consumes small space – typically a few bytes per replica – and it is stored only when the replica itself is present.

An update record is stored on disk only while the update is in progress. The space consumed by update records on a node is $(S+\alpha M/R)UD$. Here, $(S+\alpha M/R)$ shows the average space overhead of an update record on a node: $S$ is the size of *gU* and *gRetireTime*, $\alpha$ is the proportion of updates that

shrink the replica set size, $M$ is the average object size, and $R$ is the average replication factor for objects — thus, $\alpha M/R$ shows the time-averaged space overhead of *gSavedData*. $U$ is the average number of objects updated per second and $D$ is the average update lifetime, including the deletion wait period (Section 2.7) and delays introduced by retirement-aggregation (Section 5.2). In Porcupine, $S \approx 60$ bytes, $\alpha \approx 1/50$, $M \approx 5000$ bytes, $R \approx 2$, $U \approx 30$, and $D \approx 120$. Thus, total amount of stable storage used is about four hundred kilobytes.

# 7 Conclusions

We have described a new decentralized replication algorithm designed for Internet servers in this paper. Following are key features of our algorithm.

- Eventual consistency under most failure types, e.g., node and link node failures and sudden node retirements.
- Any replica can issue updates any time.
- Support for dynamic replica addition and deletion.
- Minimal space overhead, especially, efficient object deletion.
- Minimal computational and networking overhead in the common case.

As future work, we plan to investigate the space, time, and computation complexities of the algorithm under update conflicts. In addition, we are studying the implementation of semantically richer operations, e.g., multi-object transactions, on top of our algorithm.

# Acknowledgements

We thank Robert Grimm, Mike Swift, Brian Bershad, and the anonymous reviewers for giving us valuable comments that improved the quality of the paper immensely. This work was supported in part by DARPA grant F30602-97-2-0226 and NSF grant EIA-9870740.

# References

[1] Yair Amir. *Replication using group communication over a partitioned network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[2] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *USENIX Annual Technical Conference*, Anaheim, CA, 1997.

[3] Philip Bernstein and Eric Newcomer. *Principles of Tranasction Processing*. Morgan Kaufmann, 1997.

[4] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems (TOCS)*, 5(1):272–314, February 1987.

[5] Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995.

[6] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *16th ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 1–12, Santa Barbara, CA, August 1997.

[7] David K. Gifford. Weighted voting for replicated data. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 150–162, Pacific Grove, CA, 1979.

[8] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource Discovery in Distributed Networks. In *18th ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 229–237, April 1999.

[9] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems (TOCS)*, 4(1):32–53, February 1986.

[10] Peter J. Keleher. Decentralized Replicated-Object Protocols. In *18th ACM Symp. on Princ. of Distr. Computing (PODC)*, April 1999.

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

[12] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Computer Systems (TOCS)*, 9(2):125–142, 1991.

[13] Microsoft. *Windows 2000 Server Resource Kit*. Microsoft Press, 2000.

[14] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, UK, September 1994.

[15] P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *ACM SIGCOMM*, Stanford, CA, August 1988.

[16] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th Symp. on Operating Systems Principles (SOSP)*, pages 288–301, St. Malo, France, October 1997.

[17] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the Internet. Technical Report HA6177000-980305-01-TM, AT&T Labs, March 1998.

[18] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. UCLA-CSD-970044.

[19] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *17th Symp. on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[20] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study. http://www.research.digital-.com/wrl/projects/newsbench/, September 1998.

[21] Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems (TODS)*, 4(2):180–209, June 1979.

[22] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 1998.

[23] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *11th ACM Symp. on Princ. of Database Systems (PODS)*, pages 149–163, San Diego, CA, 1992.

```
public proc PushApply()
  if gU≠NULL ∧ gU.state = ACTIVE
    if gU.done=gU.target // Update done ⟨4⟩
      gU.state ← RETIRING
      gU.done ← {me}
      gSavedData ← NULL
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.target \ gU.done)
        if node ∉ gU.peers
          // Node will delete the replica.
          data ← NULL
        elif gData ≠ NULL
          // I store valid contents.
          data ← gData
        else
          data ← gSavedData
        Send(node, UpdateRequest,
             me, gU, data)

public proc UpdateRequest(caller, u, data)
  ok ← ApplyUpdate(u, data)
  Send(caller, UpdateReply,
       u.ts, ok, gU.done, gU.target)

public proc UpdateReply(ts, ok,
        done, target)
  if ¬UpdateOverwritten(ts)  // ⟨5⟩
    if ¬ ok
      gU.state ← SUSPENDED;
      return
    gU.done ← gU.done ∪ done
    gU.target ← gU.target ∪ target // ⟨6⟩

proc UpdateOverwritten(ts): bool
  return gU=NULL   // The update retired
       ∨ gU.ts > ts // A new update arrived

proc IAmCoordinator(u): bool
  return true
```

**Figure 5**. *Update propagation.* PushApply *is called periodically to push the newest update to participants.* UpdateRequest *is executed on remote nodes in response to* PushApply. UpdateReply *is called on the coordinator to handle replies from* UpdateRequest.

```
public proc PushRetire()
  if gU≠NULL ∧ gU.state = RETIRING
    if gU.done=gU.target
      gRetireTime ← Now()
      gU.state ← RETIRED
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.target \ gU.done)
        Send(node, RetireRequest, me, gU.ts)

public proc RetireRequest(caller, ts)
  if ¬UpdateOverwritten(ts)
    gRetireTime ← Now()
    gU.state ← RETIRED
    gSavedData ← NULL
  Send(caller, RetireReply, me, ts)

public proc RetireReply(node, ts)
  if ¬UpdateOverwritten(ts)
    gU.done ← gU.done ∪ {node}
public proc RemoveUpdate()
  if gU≠NULL ∧ gU.state=RETIRED
     ∧ Now() >gRetireTime+WAIT
    gU←NULL // Delete the update. ⟨7⟩

public proc MessageArrived(msg)
  if msg.ts < Now() - WAIT
    return // message too old. just ignore it
  dispatch msg
```

**Figure 6**. *Update retirement.* PushRetire *is called periodically to push retirement notices to participants.* RetireRequest *is executed on remote node in response to* PushRetire. RetireReply *is called on the coordinator to handle replies from* RetireRequest. RemoveRequest *is called periodically to remove retired updates. MessageArrived is called for every incoming message to discard messages that are too old.*