

Consistency Management in Optimistic Replication Algorithms

Yasushi Saito

June 15, 2001

Abstract

Optimistic replication algorithms allow replica contents to become stale but in a controlled way. In return, they become far more efficient and available than traditional replication algorithms that keep all the replicas consistent, especially when the network and computers are unreliable. The use of optimistic replication has grown explosively due to the proliferation of the use of the Internet and mobile computing devices, but its systematic study has begun only recently. This report surveys optimistic replication algorithms. In particular, it focuses on mechanisms for propagating updates among replicas and studies how they affect the functional characteristics of the systems, including data consistency guarantees and performance.

1 Introduction

Distributed computing systems replicate objects on multiple nodes to improve availability and performance. Replication improves availability by allowing access to the objects even when some nodes are not functional. Replication also improves performance by letting users access data from a nearby node to avoid remote network access or from an idle site to achieve a better load balance.

Replication algorithm is at the core of any replicated service, responsible for reading and updating *replicas*, or physical copies of an object. An important design issue in replication is how replicas are presented to users. Traditional *pessimistic* replication algorithms offer single-copy semantics, that is, they give users an illusion of having only a single, highly available copy of an object by keeping the replicas identical all the time [6, 17]. They are called “pessimistic” algorithms, because they prohibit accesses to a replica unless the replica contents are provably up to date. Although these algorithms are essential in a class of applications, such as banking, that must give correct answers all the time at all cost, they have one ma-

ior drawback: stringent hardware requirements. For example, the primary-copy with fail-over algorithm [6], the most popular among pessimistic replication algorithms, assumes that it can accurately distinguish a crashed node from a live but unresponsive node so that it can reliably re-elect a primary replica should it fail. Such distinction is theoretically impossible [13, 22] and probabilistically possible only by investing in redundant network hardware and over-capacity computers.

Optimistic replication algorithms allow data presented to users to become stale but in a controlled way. A key feature that separates optimistic replication algorithms from pessimistic counterparts is the way updates to objects are handled: whereas pessimistic algorithms update all the replicas at once and possibly block read requests from users during the update application, optimistic algorithms propagate updates in the background and allow any replica to be read directly most of the time. This feature makes optimistic algorithms more available and more efficient using unreliable network media and inexpensive computers. Optimistic replication is not a new idea — for example, magnetic tape backup, used for decades, is a crude form of optimistic replication — but its use has recently grown explosively due to the proliferation of the use of the Internet and mobile computing devices. Systematic treatments of optimistic algorithms, including the study of efficient update transfer and replica consistency management mechanisms, have begun only recently.

This report surveys optimistic replication algorithms. In particular, it focuses on the update propagation mechanisms invented for optimistic replication and study how these mechanisms affect the functional characteristics of the system, including data consistency guarantees and performance.

In the remainder of this section, we first overview the advantages and the applications of optimistic replication. Next, we classify optimistic replication algorithms by their update propagation strategies — where an update is issued, what is transferred as an update, and who trans-

fers an update — and discuss how the choices of the strategy affect the systems' usability and functional characteristics. Finally, we introduce two challenges optimistic replication algorithms face: replica consistency management and performance scaling.

1.1 Advantages of optimistic replication

Optimistic replication algorithms own many advantages over pessimistic algorithms by letting replica contents diverge. The key advantage is their *fault tolerance*. Optimistic algorithms work well over slow or unreliable network links, because they can propagate updates among replicas in the background without blocking accesses to any replicas. This is in contrast with pessimistic counterparts that disallow accesses to replicas until it stores the newest content.

A related advantage is the *networking flexibility*. Optimistic algorithms work well over intermittent or incomplete network links [65, 61, 25]¹ by allowing updates to be exchanged between any pair of nodes. This property is essential in a mobile environment in which nodes can communicate with each other only occasionally and unpredictably.

Some services deploy optimistic replication algorithms, because their fault tolerance and networking flexibility lets the system be built of inexpensive, failure-prone hardware and still maintain the consistency of data [67].

Finally, optimistic algorithms often improve *site autonomy* by requiring less coordination among sites. For example, some services (e.g., FTP mirroring [56]) allow a replica to be added without any administrative change on the existing replicas.

1.2 Applications of optimistic replication

Optimistic replication is particularly attractive in environments in which the communication between sites is slow and unreliable. Usenet, the wide-area bulletin board system deployed in 1979, is the oldest and still the most popular optimistically replicated service [35, 47, 69]. Usenet consists of thousands of servers connected in an ad-hoc way. Each server replicates all the news articles² so that a user can read any article from the nearest server. News

¹Incomplete links allow a replica to communicate with only a subset of peer replicas.

²In practice, each Usenet server stores only a subset of newsgroups to conserve the network bandwidth and the storage space. Still, articles in a specific newsgroup are replicated on all the servers that subscribe to the newsgroup.

articles are propagated among servers by periodic flooding along inter-server links. This periodic article propagation creates a delay before an article posted on one server reaches another, sometimes as long as a week. While this delay often confuses users, many accept it to be a reasonable cost to pay for Usenet's excellent availability.

Optimistic replication is also used to improve the performance and the availability of wide-area distributed data services, such as WWW [15, 78, 77, 37, 5, 31], FTP [56], and directory services (e.g., Xerox Clearinghouse [18], DNS [53, 53], Active Directory [49], and Grapevine [9]). Optimistic replication is attractive for them for many reasons. First, they must replicate data over unreliable long-haul network links. Second, they often operate under a tight budget. Finally, data inconsistency is inherent in these services (e.g., HTTP [21] and DNS [52, 53] explicitly allow stale data to be presented to users) and using a replication algorithm with a loose consistency guarantee does not degrade their end-to-end service quality any more.

Optimistic replication is a key enabling technology in mobile computing systems that need to replicate data on portable devices. They allow users to read or update the data while they are disconnected, merging the modifications with other nodes when they reconnect [25, 61, 65, 66]. Applications with similar demands include mobile file systems (e.g., Coda [38, 55] and Roam [65]) and mobile e-mail systems (e.g., Bayou [75] and Lotus Notes [54]).

1.3 Taxonomy of optimistic replication algorithms

In this report, we focus on how optimistic algorithms keep replicas consistent. Admittedly, there are other legitimate approaches to study optimistic replication algorithms, including the type of consistency guarantees [25] and the resolution of update conflicts [74]. We chose to focus on the mechanical aspects of the algorithms, because they have a broad impact on the systems' performance and consistency guarantees (Sections 2 and 3)

We classify optimistic replication algorithms broadly along three axes: *where* an update can be issued, *what* is transferred as an update, and *who* transfers an update.

1.3.1 Where can an update be issued?: Update transfer model

The update transfer model determines where an update can be issued and how it is propagated. **Single-master** systems statically designate one replica as the *master* that stores the authoritative copy of the object. All updates are accepted first on the master and are then propagated to other replicas, or *slaves*. **Multi-master systems** let any replica issue an update at any time, and they exchange and merge updates among replicas in the background [61, 18, 9, 2, 45, 25].

The main advantage of single-master systems is the simplicity. Because updates are accepted only at one place, single-master systems can detect and report update conflicts to users immediately, making the system less confusing to users. They are simpler algorithmically as well, because updates flow only one-way, from the master to the slaves. The disadvantage of single-master systems is that the master becomes the single point of failure.

Multi-master systems are more available than single-master systems. Their down sides, in addition to the algorithmic complexity, is the *lost-update problem*: they may lose the effects of some updates, because update conflicts are detected after the updates are accepted by replicas and the users who issued them have long logged out from the system.

1.3.2 What is transferred as an update?: The unit of transfer

A change to an object is expressed either by the new object contents or by its (often semantic) description (“log”). Systems that exchange contents are called **contents-transfer** systems, whereas systems that exchange log are called **log-transfer** systems [49, 64]. Consider a bibliography database for example. When the “author” field is updated at a replica, a contents-transfer system would transfer the entire database contents to other replicas, whereas a log-transfer system would transfer only a description of the update such as below.

```
UPDATE bibliography SET author="Herman  
Melville" WHERE title="Moby Dick"
```

Log-transfer owns several advantages over contents-transfer. First, log-transfer can handle update conflicts more flexibly, especially in multi-master systems. For example, two updates that modify the authors of two different books can be merged trivially in log-transfer systems,

but there is no choice other than either asking the user for the resolution or discarding one update and taking the other in contents-transfer systems. Second, as is evident from the example, log-transfer reduces both the computational and the networking overhead, especially when the object size is large and the update is small.

On the down side, log-transfer systems tend to be more complex than contents-transfer systems for three reasons. First, each replica must remember the history of updates in addition to the replica contents. Thus, one not needs not only complex on-disk data structures for maintaining the log, but also the algorithm for trimming the log entries for conserving space (Section 3.4.2). Second, to maintain replica consistency, the system must determine the set of updates to be sent to other replicas and the order these updates are applied. Third, because updates are described semantically, log-transfer is difficult to be implemented transparently to applications, although several systems attempt to separate an application-dependent logic from the replication algorithm itself to make the latter reusable [25, 75, 71].

1.3.3 Who transfers an update?: Direction of transfer

Direction of transfer determines which replica is responsible for transferring updates. **Pull-based** algorithms make each replica responsible for polling other replicas and downloading new updates, whereas **push-based** algorithms make a replica with a pending update responsible for delivering the update to other replicas.

Pull-based systems never send the same update to the same replica twice, because the set of updates to be received is determined precisely through polling. Thus, pull-based systems are suited in environments in which the network bandwidth is scarce [75, 65, 26]. In addition, each replica only needs to keep track only of its own state in pull-based systems (i.e., the set of update it has received), because the state of other replica is obtained through polling. This property makes pull-based systems attractive in Internet environments in which replicas are managed autonomously, e.g., in FTP mirroring [56] and Usenet article caching [39]. On the other hand, push-based systems are computationally efficient, because they obviate periodic polling and have replicas communicate only when the object is updated. Furthermore, they reduce the update propagation delay, especially in fully connected networks, by pushing updates immediately after being issued. Thus, push-transfer is attractive in sys-

tems used in fully connected environments — examples include NIS [73], Usenet [35, 69, 47], and relational database systems [16, 19, 27].

1.3.4 Summary: classifying existing optimistic replication systems

Figure 1 shows how existing optimistic replication systems are classified. In general, systems on the left side are simpler but inefficient, while systems on the right are versatile but complex.

1.4 Challenges

Optimistic replication algorithms face two main challenges: keeping replicas consistent and scaling performance. While these challenges are not unique to optimistic replication, they are complicated, because these algorithms let updates be issued at multiple replicas at the same time and optimistic algorithms often discover update conflicts long after the updates are issued.

1.4.1 Maintaining replica consistency

Maintaining the *consistency* of replica, or controlling the quality of replica contents, is the most crucial function of any replication service. By definition, optimistic replication algorithms cannot guarantee strict single-copy consistency. Thus, the first challenge here is to define the type of replica consistency an algorithm can guarantee — an implementation often follows naturally from the type of guarantee. We distinguish two types of replica consistency guarantees that a system would provide in this report: eventual consistency and view consistency.

Eventual consistency: Eventual consistency demands the consistency of the replicas in a quiescent environment. In other words, it guarantees that whatever the current state of the replicas is, if no new update is issued and the replicas can communicate freely for a long enough period, the contents of all the replicas become identical eventually.

Eventual consistency is important for two reasons. First, it is the minimal requisite of a replication algorithm; without this guarantee, the replica contents may remain corrupted forever, making the system practically useless. Second, an eventually consistent

service usually makes a best effort to disseminate updates quickly among replicas, and such a “best effort” is strong and practical enough for many applications.

Several design choices exist as to how eventual consistency is achieved when updates may be issued concurrently — for example, whether to order updates totally or partially, or whether to let the user specify the ordering or to let the system determine the ordering automatically. We discuss these issues in Section 2.1.

View consistency: Eventual consistency by itself provides little end-to-end guarantee about the quality of data — that the replica contents eventually converge give means little to users. The goal of view consistency guarantees is actually to control the quality of data by intervening on replica read (and sometimes update) requests. This guarantee is optional, and in fact, many systems serve arbitrarily stale data to users.

View consistency includes *causal consistency* that preserves partial orderings among read and write requests, and *bounded inconsistency* that explicitly limits the degree of replica inconsistency. For example of causal consistency, consider a replicated password database [9, 74]. A user may change her password on one replica and later fail to log in from another replica using the new password, because the change has not reached the latter replica. Such a problem can be avoided by having a *causal read guarantee*, that is, to make a read request honor the past updates made by the same user. An airline seat reservation system is an example in which bounded inconsistency guarantee is useful. Airline companies allow overbooking to streamline both the reservation system and the airline operations, but they must limit the degree of overbooking to avoid hurting their reputation. We discuss view consistency maintenance in more detail in Section 2.3.

1.4.2 Scalability

Optimistic replication algorithms must perform efficiently under a large workload to be practically useful. The size of the workload grows along three axes: the object size, the number of replicas, and the number of objects. These axes are mostly orthogonal — for example, some algorithms scale to many replicas well but not to large objects.

²Coda transfers contents for ordinary files and the log for directories.

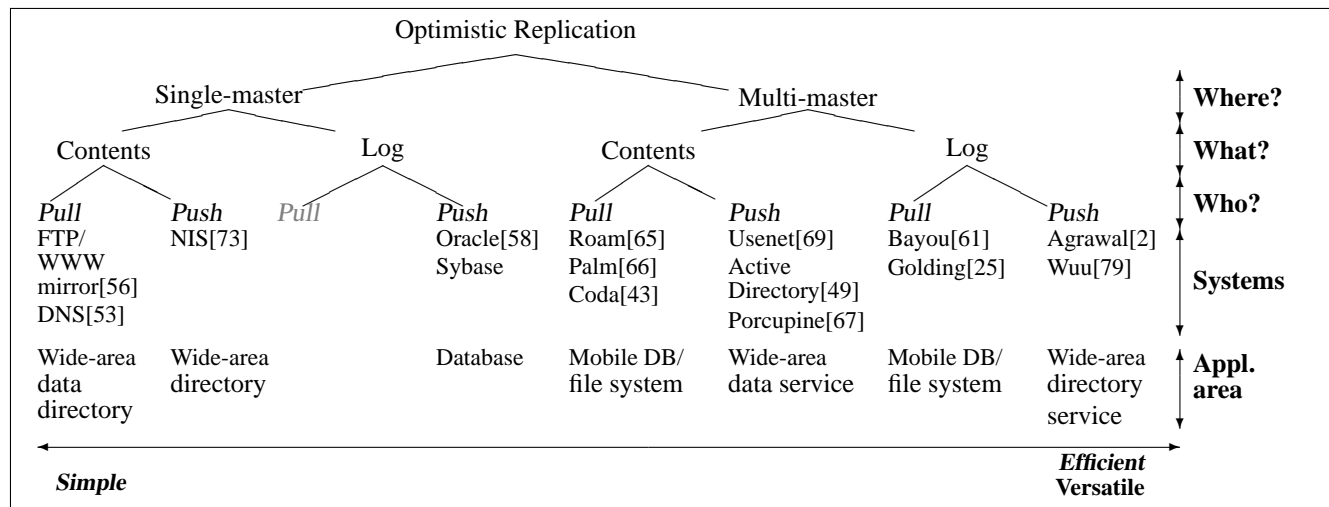


Figure 1. Classification of existing optimistic replication systems.

The space efficiency of a system is often as important as the performance. Replication algorithms must maintain several data structures as well as the object contents themselves — the log of updates, the names of other replicas, etc. These data structures should be small, and should be able to be trimmed if necessary so as to leave the maximum amount of space for what users actually want, i.e., the object contents.

1.5 Structure of the report

Section 2 surveys how optimistic algorithms handle the challenge of maintaining replica consistency. In particular, we decompose this problem into three sub-problems — distributing updates, ordering updates, and detecting and reconciling updates — and describe how various classes of algorithms solve these problems.

Section 3 discusses optimistic algorithms handle the challenge of scaling the performance. We study this issue along three dimensions — scaling to large objects, scaling to many replicas, and scaling to many objects — and also survey techniques proposed for helping system scale to large workload. We argue that the performance of optimistic replication algorithms is largely determined by their update propagation strategy. We further show that a system’s performance requirements — e.g., supporting many replicas over unreliable links — often determine the type of algorithm it uses.

Section 4 concludes the report by discussing several

open issues that have eluded satisfactory solutions.

2 Maintaining replica consistency

This section surveys the techniques for maintaining two types of replica consistency guarantees: eventual consistency guarantee and view guarantee.

2.1 Maintaining eventual consistency

This section discusses the methodologies for maintaining eventual replica consistency for various classes of update propagation strategies introduced in Section 1.3. We start with pull-transfer systems. Later in Section 2.1.4, we show that push-transfer algorithms are all derived from pull-transfer systems using one of the two techniques: blind pushing or state estimation.

The problem of maintaining eventual replica consistency is broken down into the following set of sub-problems. Algorithms introduced in this section are presented along this breakdown.

1. Distributing updates among replicas.
2. Determining the order of update application.
3. Detecting and reconciling conflicts among updates.

2.1.1 Pull-based single-master systems

Single master systems designate one replica as the master that is responsible for accepting and applying changes. Other replicas, or slaves, receive changes from the master.

Replica consistency management is trivial in single-master systems. Contents-transfer update propagation is usually implemented using a simplified variation of Thomas write rule (Section 2.1.3.1). Here, each replica stores a timestamp that shows the last time its contents were modified. A slave replica obtains the master's timestamp periodically and downloads the master replica's contents only if its timestamp is older than the master's.

Log-transfer is equally easily implemented by remembering of the last update applied on each replica.

Because updates are accepted only at the master, update ordering and conflict detection are both handled by the master replica using common database concurrency control techniques such as two-phase locking and optimistic concurrency control [29, 6].

2.1.2 Pull-based multi-master log-transfer systems

Multi-master log-transfer algorithms are the most versatile and most complicated among optimistic replication algorithms. Maintaining eventual consistency in these systems is complicated, because multiple updates may be issued simultaneously on different replicas and they may reach other replicas in an arbitrary order.

2.1.2.1 Distributing updates replicas using timestamp vectors:

Multi-master replication algorithms must ensure that each replica receives all the updates issued by all the replicas, even when the original issuers of the updates are not directly reachable. This requirement, called *any-to-any update transfer* is essential in mobile environments in which network links are usually incomplete, but it is also important in fully connected environments to avoid one node's failure stalling the update propagation of the entire system [65, 61, 25].

A naive algorithm for achieving any-to-any update transfer is to let each replica log all the updates it receives and send the entire log to other replicas either periodically or on demand. Given enough time for replicas to communicate with one another at least indirectly, all the replicas will receive all the updates. Notice that we are not concerned about how replicas actually communicate. For example, in a mobile environment, the replica communication topology would be determined by how users

meet and synchronize their devices. In many Internet services, the replica communication topology would be set up in an ad-hoc and semi-static way through human negotiation. As far as replicas can communicate with one another at least indirectly, this algorithm can distribute all the updates to all the replicas. This naive algorithm has an obvious downside, however: it becomes slower and consumes more disk space as it accumulates more updates.

Timestamp vector is a technique used in pull-based systems to minimize the number of updates exchanged between replicas [60, 25, 61, 1, 45, 20]. Each replica keeps an on-disk timestamp vector TV , an N -element array of *timestamps*, which summarizes the state of the replica. Here, N is the total number of replicas in the system and the timestamp is any number that increases monotonically — a logical clock [46], a wall clock [50, 51], or a counter that increments at each update issuance may suffice. $TV[j]$ on replica j ³ shows the time an update is last issued at replica j ; therefore, $TV[j]$ is also called replica j 's timestamp. Each update exchanged among replicas is associated with its issuer's timestamp at the moment of issuance. Other elements of TV have the following meaning: when $TV[i] = v$ on replica j , replica j has received all the updates issued at replica i and with timestamps up to v .

Figure 2 shows the update propagation procedure using timestamp vectors. To propagate updates from replica j to replica i , replica i first sends its timestamp vector, TV_i , to replica j . Replica j compares TV_i with j 's own timestamp vector, TV_j , element by element. If $TV_j[k] < TV_i[k]$ for some k , replica j sends to replica i all the updates issued at replica k , stored on j , and with timestamps larger than $TV_i[k]$. This process ensures that replica i receives all the updates stored on replica j and are absent from replica i , and that replica i does not receive any update that it has already received. After the updates are sent, TV_i on replica i is set to the pair-wise maxima of TV_i and TV_j . By swapping the roles and let replica j receive updates from replica i , the two replicas will have the same set of updates.

Timestamp vector is useful not just for update propagation. It is an efficient mechanism for estimating the state of a remote node (i.e., the number or the degree of events happened on the node). Other applications of timestamp vectors include write conflict detection (Section 2.1.2.3), update ordering (Section 2.1.2.2), and update log truncation (Section 3.4.2).

³Replicas are assumed to be named by numbers from 0 to $N - 1$.

2.1.2.2 Ordering updates: Having replicas receive the same set of updates is not enough to achieve eventual consistency in multi-master systems, because the replicas may receive the updates in a different order. The replicas must sort and apply the updates in a well-defined order. We describe two approaches for update ordering: the use of distributed protocols for informing replicas of the progress of update propagation to order updates totally, and the use of semantic relationships among updates to order updates partially.

Total update ordering: Eventual consistency can be achieved naturally by forcing all the replicas to apply updates in the same order. We overview three techniques developed to let replicas agree on such total update ordering.

One approach, proposed by Golding [26], uses *ack vectors* to learn about the status of other replicas. An *ack vector* AV is an N -element array of timestamps, exchanged between replicas and updated just like timestamp vectors. $AV[j]$ on replica j is defined to be $\min_{0 \leq i < N} TV[i]$ (i.e., j has received all the updates no newer than $AV[j]$, regardless of their issuers). Likewise, $AV[k]$ on replica j represents j 's conservative estimate of the newest update received by replica k . Thus, all the updates whose timestamps are older than $\min_{0 \leq i < N} AV[i]$ are guaranteed to have been received by all replicas, and they can be sorted by their timestamps and applied to the replica. This technique requires that the replicas' timestamps advance roughly at the same speed. Otherwise, a replica with a very slow timestamp may stall the progress of the *ack vectors* of all the replicas by keeping $\min_{0 \leq i < N} TV[i]$ from increasing. Therefore, Golding's algorithm uses loosely synchronized wall clocks [50, 51] as timestamps.

The second approach, proposed by Bayou [61], designates one replica as the *home*. The home replica unilaterally and totally orders the updates in any order it chooses. The home replica assigns a monotonically increasing commit sequence number (CSN) to each ordered update. The mappings between the updates' timestamps and their CSNs is propagated back to other replicas by piggybacking them onto ordinary update messages. Notice the difference between Bayou's home node and the master node in single-master replication systems. In Bayou, the home replica is responsible only for update ordering, and the updates can be transferred between any pair of nodes, whereas in single-master systems, the master node is responsible update issuance, ordering and

propagation.

The third approach, proposed by Deno [36], adapts the quorum consensus protocol [23, 76] to an optimistic environment. The administrator of Deno allots a *ballot* of a certain weight to each replica, with the sum of ballots held by all the replicas being 1.0. Each update is associated with a *vote* that increases by the replica's ballot when received by a replica. The application of an update is deferred until its vote exceeds 0.5. When multiple updates circulate among replicas simultaneously, only one of them will win the vote and others will be discarded silently. An update circulates among replicas up to twice in Deno — once to collect votes, and again to inform the replicas of the vote passage. This protocol totally orders updates by the order they acquire majority votes.

The three approaches have trade-offs. The advantage of the first approach is that it is decentralized and that it never aborts updates. Its downside is livelocking — a single dead replica may hamper the progress of all the other replicas — and this problem worsens as the number of replicas increases. Bayou's home-based algorithm alleviates the livelock problem, because as long as the home replica is functional, it can order the updates timely. It is also flexible, because the home replica can resolve update conflicts in any way without regard to other replicas [75]. On the other hand, the Bayou's algorithm is more complicated than Golding's. Deno supports a spectrum of the degree of centralization; assigning the entire ballot to a single replica will make Deno behave like Bayou, while assigning an equal weight to each replica will make Deno behave more like a traditional voting scheme. On the downside, Deno allows only one update to be outstanding at a time, and all other updates that fail to win majority votes are aborted. Thus, it is not suited for applications that experience large communication delay and high update frequency.

Partial update ordering: While being intuitive, total update ordering has two disadvantages. First, it may delay update application unnecessarily, because it ignores semantic relationships among updates such as update commutativity. Second, total ordering cannot preserve dependencies among updates issued at different replicas. For example, consider a source code management database, in which a library module is modified on one replica, and an application program is modified on another replica in such a way that it depends on the new library module [74]. In such a case, we want the library update to be applied before the application update on all the replicas, but we

may mix the two with updates to unrelated modules. Such a constraint cannot be expressed using total update ordering mechanisms.

Several systems address the shortcomings of total ordering by taking advantage of update commutativity [79, 63, 36, 32, 43]. For example, additions and subtractions on a numeric value or file creations in the same directory can be applied in any order to produce the same result.

The idea of commutative updates can be extended to the concept of causality. Causality is a partial ordering, defined over updates, which specifies the conditions of the form “the update X must be applied before the update Y ”. Some systems let the user specify causality explicitly [7, 8, 45]. Other systems introduce a *session*, a persistent record that encapsulates the history of object accesses by a particular user; a new update issued by the user is considered to depend causally on previous updates recorded in the user’s session [74, 24].

The earliest implementation of causality attached to each update the set of names (e.g., timestamps) of causally preceding updates, and it delayed the update application until all the causally preceding updates are applied [7]. This naive solution had a disadvantage, in that the size of an update grows unboundedly as it depends on more preceding updates. Recent systems solve this problem by compressing the preceding-updates set by grouping the updates by their issuers and picking only the newest update from each group [45, 8, 75, 61]. If a system guarantees that updates issued by the same replica are propagated in the order of issuance, as is the case with timestamp vectors, this dependency compression algorithm achieves the same effect as the earlier naive solution.

2.1.2.3 Detecting and reconciling conflicting updates: Two updates conflict when a replica issues an update before it receives another update that is already circulating among replicas⁴. This situation is easily detected using timestamp vectors as follows. Remember that when one replica’s timestamp vector dominates⁵ another replica’s, the former is strictly newer than the latter in the sense that the former has received all the updates that the latter has. To detect update conflicts, the sys-

⁴This report defines update conflicts syntactically. For example, in the room reservation example, concurrent reservation requests to different rooms do conflict by this definition, but they do not in the common sense.

⁵One timestamp dominates another when each element of the former timestamp is no older than the latter’s.

tem attaches to each update the issuer’s timestamp vector at the moment of issuance (notice that in the original timestamp vector algorithm described in Section 2.1.2.1 and Figure 2, the system attaches the issuer’s timestamp to each update) [60]. When one update’s timestamp vector dominates another’s, the two updates do not conflict, because the former is issued after its issuer received the latter update. Otherwise, the two updates conflict.

Update conflicts, by themselves, do not affect eventual consistency of the object — conflict detection is purely for user convenience. Consider a conference room reservation system for example [75]. If two people submit reservation requests for the same room for the same time slot, the system can easily maintain eventual consistency by accepting one request and ignoring the other. If the people were informed about the conflict, however, they would be able to negotiate the schedule.

Resolving conflicts is fundamentally application-dependent [75, 71]. Some conflicts, such as creating two different files in the same directory [43], can be resolved simply by merging them, but others, such as the previous room reservation example, need to be resolved manually.

2.1.3 Pull-based multi-master contents-transfer systems

This section discusses how multi-master contents-transfer systems maintain eventual consistency. To propagate and order updates, contents-transfer systems need to agree only on which replica is the newest — transferring the contents of the newest replica to others will bring all the replicas up to date. We first introduce Thomas write rule, a simple protocol used to determine the newest replica contents. Next, we describe algorithms used by applications that must detect update conflicts.

2.1.3.1 Thomas write rule: *Thomas write rule* gets its name from the seminal paper that introduced an algorithm for pessimistic, voting-based replication [76], but it is now more widely used in optimistic replication systems. It associates with each replica a timestamp — usually a loosely synchronized wall clock [50, 51], but sometimes a counter (e.g., Lotus Notes [64]) — that shows the last time the replica is modified [76, 9, 53, 64, 30]. Each replica obtains a peer replica’s timestamp either periodically or on demand, and it downloads the peer’s contents only when its timestamp is older than the peer’s. Given enough time for replicas to communicate with one another at least indirectly, all the replicas will converge on

the newest contents.

Object deletion requires a special treatment using Thomas' write rule, because the timestamp cannot be read from a deleted replica. Three solutions have been proposed to solve this problem. The first solution is simply to require a human intervention to delete objects (e.g., DNS [53] and NIS [73]). The second solution is to keep the timestamps (but not the contents) of deleted replicas on disk. These timestamps are also called "tombstones" [49] or "death certificates" [18]. The third solution, proposed by the Porcupine mail server [68, 67], propagates an update in two phases. In the first phase, the update is disseminated among replicas. The update record, logged on disk separately from the replica contents, acts as a tombstone when the update deletes the object. After all the replicas confirm the reception of the update, a separate retirement notices are circulated among the replicas to delete the update record.

2.1.3.2 Detecting update conflicts: Thomas write rule has one shortcoming: all conflicting updates are silently discarded except for the newest one. Systems designed for multi-user collaboration, such as Coda [38, 44], Roam [65] and Locus [60], need to know when the updates conflict and notify the users. These applications combine timestamp vectors and contents-transfer. They work almost like the systems based on Thomas write rule, except that they associate with each replica a timestamp vector, TV , instead of a timestamp. As before, $TV[i]$ on replica i shows the last time an update was issued at i . Other elements of TV are updated by merging them with other replicas' timestamp vectors (Section 2.1.2.1). Timestamp vectors defined this way can detect update conflicts using the same technique introduced in Section 2.1.2.3.

Several systems offer an alternative solution by naturally extending Thomas write rule [28, 58, 64, 66]. In this scheme, each update is associated with the issuing node's timestamp just before the update issuance. When the receiver-side replica's timestamp differs from the one attached to the update, the update is considered to conflict with the one previously applied on the receiver-side replica. While being simpler than the technique based on timestamp vectors, this technique may report a false-positive conflict when the receiver-side replica misses older updates.

2.1.4 Push-transfer systems

Push-transfer systems are actually derivatives of pull-based counterparts. Pull- and push-based systems differ only in the way updates are distributed. Other issues, including update ordering, conflict detection and reconciliation, can be addressed using the same techniques developed for pull-based systems.

The pull-transfer algorithms discussed so far associate with each replica a timestamp or something similar (e.g., a timestamp vector) to represent the state of the replica. They then let each replica retrieve a peer replica's timestamp and discover the set of updates or the contents to be downloaded. On the other hand, push-transfer systems must identify the set of updates to be sent to another replica without a contact. This can be achieved in two ways: by blindly pushing all the new updates or by estimating the state of other replicas.

2.1.4.1 Blind pushing: In the blind-pushing scheme, a replica with an update blindly delivers the update to all the replicas that are communicable. This approach is simple and effective in services that connect replicas in such a way that an update flows through more or less a fixed path and is unlikely to be received by the same replica twice. Single-master systems, Usenet [69], relational database systems [58], and the Porcupine mail server [68] are the examples of the services that use blind pushing. The receiver-side replica uses one of the pull-transfer mechanisms (e.g., Thomas write rule in Usenet and Porcupine) to filter out duplicate updates.

2.1.4.2 State estimation techniques: Blind pushing has an obvious downside, that is, it sends duplicate updates when each replica communicates with many other replicas [18]. This problem can be alleviated by letting each replica estimate the state of others and push only those updates that are likely to be missing on a remote replica.

An example of state-estimating techniques is *timestamp matrices*, used in multi-master, log-transfer, push-based systems [79, 48, 2]. A timestamp matrix TM , stored on each replica, is an $N \times N$ matrix of timestamps (N is the number of replicas, and the timestamp is defined as in Section 2.1.2.1). $TM[i]$ on replica i holds i 's timestamp vector, and the other rows of TM shows the replica's conservative estimate of the timestamp vectors of other replicas. Put another way, if $TM[k][j] = c$ on some replica l , then l knows that replica k at least has received all the

updates issued at replica j with timestamps up to c . The update propagation process, Figure 3, is similar to the one using timestamp vectors (Figure 2). The only difference is that when sending updates to replica j , the sender uses its $TM[j]$ as an estimate of replica j 's timestamp vector rather than receiving the vector from replica j . The receiver-side replica's timestamp matrix is updated by computing the pairwise maxima of itself and the sender's matrix, piggybacked on the update message. Timestamp matrix reduces the number of duplicate updates, but it still cannot totally eliminate duplicate updates completely — e.g., if two replicas receive an update simultaneously, they may send the update to the same replica twice.

Thomas write rule can also be extended naturally to become a push-based algorithm using a similar technique: let each node maintain a vector of timestamps TMV that conservatively estimates the timestamp of other replicas. Replica i sends its contents to another replica j only when $TMV[i] > TMV[j]$ on replica i . Active Directory [49], a replicated hierarchical directory service provided by Windows 2000, uses an algorithm similar to the above. The difference is that Active Directory replicates many objects on a site (each node in the name space tree is an object), and the vector actually estimates the largest timestamp among all the objects replicated on a site.

2.2 Trading off replica consistency and update application delay

The algorithms described so far delay applying updates until their final application order is determined. Thus, slower the communication among replicas is, or more replicas there are, longer the update application is delayed. This problem is alleviated by applying updates *tentatively* as soon as they arrive [75]. When the tentative order turns out to be different from the final order, these updates are undone and re-applied in the correct order, if necessary. Tentative updates allow the user to see more up-to-date contents at the price of an occasional confusion when they are undone and re-applied. Tentative updates are used also in single-master systems to let the user update a slave replica while being disconnected from the master [28, 38].

Tentative updates pose two implementation challenges. First, the updates need to be able to be undone, which is not always possible especially when they interact with the real world (e.g., firing a missile or paying money). Second problem, applicable only to multi-master systems, is that the number of undos and redos increases at $O(N^2)$ (N is

the number of replicas), resulting in a large computational cost (Section 3.2.2).

2.3 Managing view consistency

Optimistic replication algorithms, by definition, cannot ensure strict replica consistency. Eventual consistency only guarantees replica consistency in hypothetical quiescent state, and it offers little clue to users regarding the quality of replica contents. Some replicated data services do fine with just eventual consistency guarantee, because of the semantic limitations of the network protocols they use. The Porcupine mail server [68, 67] and Usenet [69] are examples of such services; replica inconsistency is no worse than potential problems caused by SMTP (for email) and NNTP (for Usenet), such as the delay of delivery and duplicate messages. On the other hand, in many services, such as the replicated password database example illustrated in Section 1.4.1, weaker forms of quality-of-data guarantees provide a visible benefit to users. This section overviews proposals for weaker replica consistency guarantees and their implementations.

2.3.1 Causal consistency

Preserving causal orderings among accesses is especially helpful in avoiding consistency anomalies that confuse users. Four types of causal consistency guarantees have been proposed depending on how reads and writes interact: “read your writes”, “monotonic reads”, “writes follow reads”, and “monotonic writes” [74]. Among them, writes follow reads and monotonic writes are actually forms of causal write consistencies, already covered in Section 2.1.2.2. “Read your writes” guarantees that the contents read from a replica always incorporate previous writes (by the same user). The password example illustrated in Section 1.4 is an example in which the read-your-writes guarantee is beneficial. “Monotonic reads” guarantees that two successive reads (by the same user) return increasingly up-to-date contents.

Both guarantees are trivial to provide when the user accesses only a single replica, but they become tricky when the user roams over multiple replicas. View consistency in roaming environments is implemented by the same technique used for maintaining causal update consistency (Section 2.1.2.2) [7, 8, 45, 74, 24]. Here, a read request is associated with (the compressed representation of) the set of causally preceding read or update requests. A request that does not satisfy its causality constraint ei-

ther is rejected outright [74], or is delayed until the replica receives enough updates to satisfy the constraint [45].

2.3.2 Bounded consistency

Several systems let the user specify the maximum degree of inconsistency allowed during a replica access.

Many systems (e.g., quasi copies [3], DNS [53], and WWW [15, 78, 77]) measure inconsistency by real-time: they guarantee that the newest object contents are propagated to the replicas within a fixed period. The advantage of this approach is the simplicity — a combination of single-master replication and periodic polling/pushing naturally achieves this guarantee. The downside is that it is practical only in single-master replication systems with highly reliable network links.

A different line of studies suggests specifying the amount of inconsistency in terms of the number of updates that the read can overlap with [63], or in terms of semantic metrics, for example, within five dollars from the amount in a bank account [40, 41, 57, 81]. They have been used to improve the performance of non-replicated database systems by increasing locking concurrency, but recent studies suggest applying them to replication both to improve performance and to enrich consistency semantics [63, 11, 62]. Earlier work in this area is implemented by a relaxed two-phase locking in which reads and writes to the same datum are granted under certain conditions and are thus often not applicable to mobile or wide-area-distributed applications designed to work over intermittent network links. A recent study [81] suggests the use of forced pushing and pulling of updates among replicas to ensure that the degree of inconsistency among replicas is bounded.

3 Scaling optimistic replication systems

This section studies the performance and the space efficiency of optimistic replication algorithms. We first discuss the computational and the networking overhead of the systems along the three axes of workload scaling: large objects, large number of replicas, and large number of objects. We next discuss the storage requirement of the systems and their efforts to curtail storage consumption.

3.1 Scaling to large objects

Scaling to large objects is determined by *what* is transferred as an update (Section 1.3.2).

Log-transfer systems support large objects better, because their cost depends on the frequency and the size of updates, but not on the size of the object.

On the other hand, the cost of contents-transfer systems increases linearly with the object size. Two techniques have been proposed to alleviate this problem without losing the simplicity of contents-transfer systems. The optimistic delta technique [5, 68], originally developed for WWW latency reduction, speculatively sends a difference between new and old object contents and falls back to a full contents transfer when the remote replica contents do not match the sender's. This technique saves the network bandwidth but not the computational cost. Another technique is to divide an object into a hierarchically structured set of subobjects and use Thomas write rule at each level of hierarchy (e.g., Rabinovich's algorithm [64], Lotus Notes [64], Active Directory [49]). This technique can be extended to allow a site to replicate just a sub-tree within the hierarchy to reduce the cost of replication further (e.g., Roam [65]).

3.2 Scaling to a large number of replicas

A system's ability to support many replicas is determined by the update transfer model described (Section 1.3.1). Supporting many replicas presents different problems for single-master and multi-master systems.

3.2.1 Propagation delay and load imbalance

Having many replicas increases the update propagation delay and sometimes creates a load imbalance among replicas. This problem is especially serious in single-master systems in which the master replica propagates updates to all the slaves. This problem, fortunately, is alleviated easily. One solution is to organize the replicas in a tree form, place the master at root, and let updates ripple down the tree from the root [15, 80, 1, 34]. It cuts the propagation delay from $O(N)$ to $O(\log N)$ (N is the number of replicas) and reduces the load on the master (and all other replicas) to a constant level. Multi-master systems such as Roam [65], Usenet [69] and Active Directory [49] can push this idea further and connect replicas in a tree structure supplemented by short-cut paths to improve the availability and the update propagation speed. Another technique employs an unreliable multicast protocol (e.g.,

Digital fountain [10]) to distribute updates efficiently in the common case and to back it up with a slower but reliable algorithm like those described in Section 2 [18, 47].

3.2.2 Increased update conflicts

Gray [28] argues that multi-master replication algorithms cannot scale to large number of replicas, because they would experience $O(N^2)$ update conflicts (Section 2.1.2.3), whereas single-master systems would experience only $O(N)$ update conflicts (N is the number of replicas). In addition to confusing users, increased update conflicts leads to increased undos and redos of updates (and thus larger computational overhead) when the system supports tentative update application (Section 2.2).

Gray makes one important assumption to derive his conclusion: each data item is updated equiprobably by all replicas. This assumption does not hold universally — for example, write sharing in UNIX file systems is known to be rare [59, 4] — and it is still not clear how severe this scalability problem is in practice. Assuming equiprobable data access, however, this problem is inherent to multi-master systems, and no satisfactory solution has been found yet. Proposed remedies include a single-master replication with a support for tentative updates [28, 55], or a contents-transfer mechanism that can resolve update conflicts efficiently (i.e., by simple overwriting) [69].

3.3 Scaling to large number of objects

While most existing optimistic replication systems are designed to manage a single object per node, several reasons exist for replicating many objects on a node. One reason is to replicate just a portion of a large database to conserve disk space [65, 69]; a natural solution is to divide the database into small pieces and replicate them selectively. Another reason, demonstrated by the Porcupine mail server [67], is to “mix and match” replicas from many objects on a node to mask the system’s heterogeneity — e.g., differences in the disk capacity and the CPU speed — and achieve a better load balance.

In theory, any single-object replication algorithm described so far can be made to support multiple objects simply by running many instances of the algorithm in parallel. However, a technique that is cheap for a single object may become prohibitively expensive for many objects. Periodic polling performed by pull-based systems is an example. Because the number of polls per node in pull-

based systems increases at $O(NX)$ (N is the mean number of replicas per object and X is the number of replicas stored on a node), pull-based systems become expensive when a node replicates many objects. Therefore, most systems that support many objects — e.g., Active directory [49], Usenet [69], and the Porcupine mail server [67] — use push-based algorithms.

3.4 Controlling the space overhead

Replication algorithms keep several types of data structures on disk in addition to the object contents to maintain replica consistency. Examples include timestamp vectors or matrices to keep track of the state of replicas (Sections 2.1.2.1 and 2.1.4), tombstones to keep track of deleted objects (Section 2.1.3.1), and the update log (Section 2.1.2.1). This section examines the space overhead of these data structures and overviews the strategies for reducing the overhead.

3.4.1 Timestamp overhead

Table 1 summarizes the amount of space occupied by timestamp-related data structures per replica for various optimistic replication algorithms. The overhead due to timestamps is usually negligible even for timestamp matrices, because each timestamp occupies just 32 or 64 bits of disk space. Nevertheless, one study proposes reducing the overhead of timestamp matrices further by collapsing the rows corresponding to the remote replicas into one by taking column-wise minimum [33]. This scheme, however, reduces the accuracy of estimation of state of remote nodes, potentially resulting in more duplicate update transfers.

3.4.2 Controlling the size of update log and tombstones

Log-transfer systems keep the log of updates that grows indefinitely as updates accumulate over time. Several techniques are proposed to trim the size of the update log. The first technique deletes all the updates older than a predefined period (e.g., one month) or uses a fixed-size update log and rolls over old updates when the log fills up [42, 70]. The second technique is based on an observation that the log trimming problem and the total update ordering problem can both be solved by discovering whether an update is received by all the replicas. Therefore, some systems use a variant of the ack vector algorithm (Section 2.1.2.2) to trim the update log [25, 65, 14].

Algorithm types	Space	Notes
Single-master pull	1	
Single-master push	N or 0	N on master, 0 on slaves.
Multi-master pull contents-transfer	1	Thomas write rule
Multi-master push contents-transfer	1 N	Blind pushing (Porcupine, Usenet) State estimation (Active Directory)
Multi-master pull log-transfer	N	Timestamp vector
Multi-master push log-transfer	N N^2	Blind pushing Timestamp matrix

Table 1. The amount of the disk space occupied by timestamps by various replication algorithms. The unit is the number of timestamps. N represents the total number of replicas in the system.

Contents-transfer systems keep tombstones for deleted objects (Section 2.1.3) that also accumulate as more objects are deleted. A tombstone, which is just a timestamp, takes far less space than an update log, which contains the entire history of changes to a replica. For this reason, and also to keep the system simple, most contents-transfer systems delete old tombstones simply by waiting a fixed period [18, 49]. Porcupine email server [68] exchanges “retire” messages among replicas immediately after the update propagation is finished to delete tombstones quickly.

3.5 Summary

Table 3.5 summarizes how the design of an optimistic replication algorithm affects their scalability. The table shows that each axis in update propagation strategy affects the scalability and the space overhead in a different way. In other words, the performance demand of an application will more or less dictate the replication algorithm to be used in the application. For example, a mobile database system that replicates a single large database in many replicas should pick log-transfer, pull-based algorithms, which is actually the case.

4 Open Issues

4.1 Improving the Performance of Optimistic Replication Algorithms

Optimistic replication algorithms have traditionally been used in mobile or widely distributed environments in which network bandwidth is the primary bottleneck. Consequently, many of them have focused on minimizing the

networking overhead and ignored the CPU and the disk overheads. Several recent systems use optimistic replication in local area network environments to make the system highly fault tolerant using inexpensive hardware — e.g., Porcupine [67] and Active Directory [49]. Such systems should save CPU cycles and disk accesses rather than network bandwidth. This issue can be addressed in three ways.

The first approach is to combine two algorithms, an efficient algorithm for the common case and a slower but fault-tolerant algorithm for the emergency case. Examples include the use of unreliable multicast protocols [18, 47] (Section 3.2.1), and the combination of distributed transactions and optimistic replication [17]. This line of algorithms, however, has failed to gain popularity due to its complexity.

The second approach, which has yet to see an investigation, is optimizing existing optimistic algorithms. For example, algorithms based on timestamp vectors use an update log that must support a complicated set of operations (e.g., retrieve updates by issuer, sort by timestamp, delete arbitrary set of updates). As another example, tentative updates (Section 2.2) require complex supports from the system to undo and redo operations. Efficient implementation of these operations is an issue largely ignored until now.

The final approach is to design new optimistic algorithms with focus on CPU/disk cost reduction. We have seen activities in this area only recently — e.g., Rabinovich’s hierarchical object decomposition [64], and Porcupine’s selective replication [67].

	Scaling			Space
	large object	many replicas	many objects	Overhead
Single master		$O(N)$		
Multi master		$O(N^2)$		
Push		$N(\text{dup update})$	Y	
Pull		Y	N	
Contents transfer	N			tombstones
Log transfer	Y			update log

Table 2. This table shows how the architecture of systems affect the performance. Shaded entries show comparatively better choices.

4.2 Tolerating wider varieties of failures

Most of the optimistic replication algorithms make several implicit assumptions about failures: a failure cures quickly, and it does not cause a permanent damage to a replica state. In practice, however, these assumptions often do not hold. Disk fill-up, disk crash, and a theft are examples of failures not handled well by existing systems. For example, algorithms based on timestamp vectors or timestamp matrices offer no way to delete a vector (or matrix) entry without the replica corresponding to the entry explicitly circulating retirement notices to other replicas [26, 61]. If a replica crashes and never recovers, all other replicas' timestamp vectors (or matrices) will contain a garbage entry that can be removed only manually. For another example, the update log in many log-transfer systems will fill up if one replica remains incommunicable for long.

This problem is a weaker variation of the Byzantine consensus problem, an active area of study. However, existing solutions to Byzantine consensus, e.g., [12], are not applicable for our purpose, because they use fast, fully connected network to perform lock-step replica synchronization.

Several systems have solved a part of this problem in ad-hoc ways. For example, Bayou allows the update log to be erased at any time to save disk space [61] and can contain limited of types corrupt replicas [72]. For another example, Porcupine [68] allows a node to retire suddenly without any announcement.

References

[1] Noha Adly. *Management of Replicated Data in Large Scale Systems*. PhD thesis, Corpus Cristi College, University of Cambridge, August 1995.

[2] Divyakant Agrawal, Amr El Abbadi, and R. C. Steike. Epidemic algorithms in replicated databases. In *16th ACM Symp. on Princ. of Database Systems (PODS)*, pages 161–172, Tucson, AZ, May 1997.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval systems. *ACM Trans. on Database Systems (TODS)*, 15(3):359–384, September 1990.

[4] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 198–212, Pacific Grove, CA, October 1991.

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

[6] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

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

[8] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems (TOCS)*, 9(3):47–76, August 1991.

[9] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4), February 1984.

[10] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *ACM SIGCOMM*, pages 56–67, Vancouver, BC, August 1998.

[11] John Carter, Anand Ranganathan, and Sai Susarla. Khazana: An infrastructure for building distributed services. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[12] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Symposium on Operating System*

- Design and Implementation (OSDI)*, New Orleans, LA, February 1999.
- [13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [14] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems (TOCS)*, 3(1):63–75, 1985.
- [15] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *USENIX Winter Technical Conference*, January 1996.
- [16] Dean Daniels, Lip Boon Doo, Alan Downing, Curtis Elsbernd, Gary Hallmark, Sandeep Jain, Bob Jenkins, Peter Lim, Gordon Smith, Benny Souder, and Jim Stamos. Oracle’s symmetric replication technology and implications for application design. In *ACM SIGMOD International Conference on Management of Data*, page 467, Minneapolis, MN, May 1994.
- [17] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.
- [18] 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.
- [19] Daniel J. Dietterich. DEC data distributor: for data replication and data warehousing. In *ACM SIGMOD International Conference on Management of Data*, page 468, Minneapolis, MN, May 1994. ACM.
- [20] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually Serializable Data Services. In *15th ACM Symp. on Princ. of Distr. Computing (PODC)*, Philadelphia, PA, May 1996.
- [21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext transfer protocol – HTTP/1.1. <http://info.internet.isi.edu/in-notes/rfc/files/rfc2616.txt>, June 1999.
- [22] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [23] David K. Gifford. Weighted voting for replicated data. In *13th Symp. on Operating Systems Principles (SOSP)*, pages 150–162, Pacific Grove, CA, 1979.
- [24] Ashvin Goel, Calton Pu, and Gerald Popek. View consistency for optimistic replication. In *17th IEEE Symposium on Reliable Distributed Systems*, October 1998.
- [25] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, UC Santa Cruz, December 1992.
- [26] Richard A. Golding. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, UC Santa Cruz, February 1993.
- [27] Alex Gorelik, Yongdong Wang, and Mark Deppe. Sybase replication server. In *ACM SIGMOD International Conference on Management of Data*, page 469, Minneapolis, MN, May 1994.
- [28] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. Dangers of replication and a solution. In *ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.
- [29] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.
- [30] James Gwertzman and Margo Seltzer. The case for geographical pushcaching. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 1995.
- [31] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *USENIX Winter Technical Conference*, February 1996.
- [32] Brad Hammond. Merge replication in Microsoft’s SQL Server 7.0. In *ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, May 1999.
- [33] Abdelsalam Heddaya, Meichun Hsu, and William E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1–3):35–57, 1989.
- [34] T. Johnson and K. Jeong. Hierarchical matrix timestamps for scalable update propagation. <http://w4.lns.cornell.edu/~jeong/index-directory/hmt-paper.ps>, June 1996.
- [35] Brian Kantor and Phil Rapsey. RFC977: Network news transfer protocol. <http://info.internet.isi.edu/in-notes/rfc/files/rfc977.txt>, February 1986.
- [36] Peter J. Keleher. Decentralized Replicated-Object Protocols. In *18th ACM Symp. on Princ. of Distr. Computing (PODC)*, April 1999.
- [37] Anne-Marie Kermarrec, Ihor Kuz, Marten van Steen, and Andrew S. Tanenbaum. A Framework for consistent, replicated web objects. Technical Report IR431, Vrije Universtat, May 1997.
- [38] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Computer Systems (TOCS)*, 10(5), February 1992.
- [39] Cornelius Krasel. *Leafnode: an NNTP server for a small sites*, 2000. <http://www.leafnode.org>.
- [40] Akhil Kumar and Michael Stonebraker. Semantic based transaction management techniques for replicated data. In *ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 1988.
- [41] Akhil Kumar and Michael Stonebraker. An analysis of borrowing policies for escrow transactions in a replicated environment. In *IEEE Sixth International Conference on Data Engineering*, pages 446–454, 1990.
- [42] P Kumar. *Mitgating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, Carnegie Mellon University, December 1994. CMU-CS-94-215.

- [43] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *Second International Conference on Parallel and Distributed Information Systems*, January 1993.
- [44] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter Technical Conference*, New Orleans, LA, January 1995.
- [45] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Computer Systems (TOCS)*, 10(4):360–391, 1992.
- [46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [47] Kurt Lidl, Josh Osborne, and Joseph Malcolm. Drinking from the firehose: Multicast USENET news. In *USENIX Winter Technical Conference*, 1994. <ftp://ftp.uu.net/networking/news/muse/usenix-muse.ps.gz>.
- [48] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 216–226, 1989.
- [49] Microsoft. *Windows 2000 Server Resource Kit*. Microsoft Press, 2000.
- [50] David L. Mills. RFC1305: Network time protocol (version 3). <http://info.internet.isi.edu/in-notes/rfc/files/rfc1305.txt>, March 1992.
- [51] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, UK, September 1994.
- [52] P. V. Mockapetris. RFC1035: Domain names — implementation and specification. <http://info.internet.isi.edu/in-notes/rfc/files/rfc1035.txt>, November 1987.
- [53] P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *ACM SIGCOMM*, Stanford, CA, August 1988.
- [54] C. Mohan. A database perspective on Lotus Domino/Notes. In *ACM SIGMOD International Conference on Management of Data*, page 507, Philadelphia, PA, May 1999.
- [55] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 143–155, Copper Mountain, CO, December 1995.
- [56] Ikuo Nakagawa. Ftpmirror – mirroring directory hierarchy with ftp. <http://noc.intec.co.jp/ftpmirror.html>, November 1996.
- [57] Patrick E. O’Neil. The escrow transactional method. *ACM Trans. on Database Systems (TODS)*, 11(4):405–430, 1986.
- [58] Oracle. *Oracle7 Server Distributed Systems Manual*, Vol. 2, 1996.
- [59] John K. Ousterhout, Herv Da Costa, David Harrison, John A. Kunze, Michael D. Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *10th Symp. on Operating Systems Principles (SOSP)*, pages 15–24, 1985.
- [60] D. Scott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency on distributed systems. *IEEE Trans. on Software Engineering*, SE-9(3):240–247, 1983.
- [61] 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.
- [62] Calton Pu, Wenwey Hseush, Gail E. Kaiser, Kun-Lung Wu, and Philip S. Yu. Divergence control for distributed database systems. *Distributed and Parallel Databases*, January 1995.
- [63] Calton Pu and Avraham Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, CO, May 1991.
- [64] Michael Rabinovich, Narain H. Gehani, and Alex Kononov. Efficient Update Propagation in Epidemic Replicated Databases. In *International Conference on Extending Database Technology (EDBT)*, pages 207–222, Avignon, France, March 1996.
- [65] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. UCLA-CSD-970044.
- [66] Neil Rhodes and Julie McKeehan. *Palm Programming: The Developer’s Guide*. O’Reilly, December 1998.
- [67] 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.
- [68] Yasushi Saito and Henry M. Levy. Optimistic replication for Internet data services. In *14th International Conference on Distributed Computing (DISC)*, Toledo, Spain, October 2000.
- [69] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study. <http://www.research.digital.com/wrl/projects/newsbench/>, September 1998.
- [70] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, 13(1):39–47, 1987.
- [71] Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop on “Beyond the PC: New Challenges for the Operating System”*, Kolding, Denmark, September 2000.

- [72] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *MobiCom*, Budapest, Hungary, September 1997.
- [73] Sun Microsystems. Sun directory services 3.1 administration guide. <http://www.sun.com/sims/Docs-4.0/html/sunds/admin/>, July 1998.
- [74] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Austin, TX, September 1994.
- [75] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th Symp. on Operating Systems Principles (SOSP)*, pages 172–183, Copper Mountain, CO, December 1995.
- [76] 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.
- [77] Duane Wessels and K. Claffy. RFC2186: Internet cache protocol. <http://info.internet.isi.edu/in-notes/rfc/files/rfc2186.txt>, September 1997.
- [78] Alastair Wolman, Geoff Voelker, Nitin Sharma, Neil Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *17th Symp. on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.
- [79] Gene T. J. Wu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *3rd ACM Symp. on Princ. of Distr. Computing (PODC)*, pages 233–242, Vancouver, Canada, August 1984.
- [80] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical cache consistency in a WAN. In *2nd USENIX Symp. on Internet Technologies and Systems*, pages 13–24, Boulder, CO, October 1999.
- [81] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th Symposium on Operating System Design and Implementation (OSDI)*, pages 305–318, San Diego, CA, October 2000.

```

type Update = record
  // The replica that issued the update.
  issuer: ReplicaID;
  // The timestamp at the moment of issuance.
  ts: Timestamp;
  // Actual update contents
  stuff: UpdateContents;
end;

// The set of update the replica has received.
var log: Set(Update)
// The replica's timestamp vector.
var tv: array [0 .. N-1] of Timestamp;

proc Reconcile(peer) // Send updates to peer
  peerTV ← Receive peer's tv.
  for i ∈ {0 .. N-1}
    if peerTV[i] < tv[i] then
      upd ← { u ∈ log such that
              u.issuer = i ∧ u.timestamp > tv_peer[i]}
      Send upd to peer.
  tv ← PairwiseMax(tv, peerTV)

proc receive_update(upd) // Called via reconcile().
  foreach u ∈ upd
    Apply u on the replica;
    tv[i] ← max(tv[u.issuer], u.ts);
  log ← log ∪ upd

```

Figure 2. Replica reconciliation using timestamp vectors

A Description of basic algorithms

```

// The set of update the replica has received.
var log: Set(Update)
// The replica's timestamp matrix.
var tm: array [0 .. N-1][0 .. N-1] of Timestamp;

proc SendUpdate(peer) // Send updates to peer.
  if all entries in tm[me] no larger than tm[peer] then
    return;

    for i ∈ {0 .. N-1}
      if tm[peer][i] < tm[me][i] then
        upd ← { u ∈ log such that
          u.issuer = i ∧ u.timestamp > tm[peer][i]}
        Send upd to peer.
        tm[peer][i] := tm[me][i]
    Send tm to peer.
    peerTM ← Receive tm from peer.
    tm ← PairwiseMax(tm, peerTM)

// Called from SendUpdate
proc ReceiveUpdate(upd)
  foreach u ∈ upd
    Apply u on the replica
    tm[me][u.issuer] ← max(tm[me][u.issuer], u.ts)
  log ← log ∪ upd

```

Figure 3. Replica reconciliation using timestamp matrices