# Optimistic Replication

Yasushi Saito, Marc Shapiro

# Optimistic replication

Yasushi Saito, Hewlett-Packard Laboratories, Alto, CA (USA),
and
Marc Shapiro, Microsoft Research Ltd., Cambridge (UK)

September 2003

Technical Report
MSR-TR-2003-60

Data replication is a key technology in distributed data sharing systems, enabling higher availability and performance. This paper surveys optimistic replication algorithms that allow replica contents to diverge in the short term, in order to support concurrent work practices and to tolerate failures in low-quality communication links. The importance of such techniques is increasing as collaboration through wide-area and mobile networks becomes popular. Optimistic replication techniques are different from traditional "pessimistic" ones. Instead of synchronous replica coordination, an optimistic algorithm propagates changes in the background, discovers conflicts after they happen and reaches agreement on the final contents incrementally. We explore the solution space for optimistic replication algorithms. This paper identifies key challenges facing optimistic replication systems — ordering operations, detecting and resolving conflicts, propagating changes efficiently, and bounding replica divergence — and provides a comprehensive survey of techniques developed for addressing these challenges.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Contents

This work is supported in part by DARPA Grant F30602-97-2-0226 and National Science Foundation Grant # EIA-9870740.

Authors' addresses: Yasushi Saito, Hewlett-Packard Laboratories, 1501 Page Mill Rd, MS 1U-34, Palo Alto, CA, 93403, USA. mailto:yasushi@cs.washington.edu, http://www.hpl.hp.com/personal/Yasushi_Saito. Marc Shapiro, Microsoft Research Ltd., 7 J J Thomson Ave, Cambridge CB3 0FB, United Kingdom. mailto:Marc.Shapiro@acm.org, http://www-sor.inria.fr/~shapiro/.

## 1. INTRODUCTION

Data replication consists of maintaining multiple copies of critical data, called *replicas*, on separate computers. It is a critical enabling technology of distributed services, improving both their availability and performance. Availability is improved by allowing access to the data even when some of the replicas are unavailable. Performance improvements concern reduced latency, which improves by letting users access nearby replicas and avoiding remote network access, and increased throughput, by letting multiple computers serve the data.

This paper surveys optimistic replication algorithms. Compared to traditional "pessimistic" techniques, optimistic replication promises higher availability and performance, but lets replicas temporarily diverge and lets users see inconsistent data. The remainder of this introduction overviews the concept of optimistic replication, defines its basic elements, and compares it to traditional replication techniques.

### 1.1 Traditional replication techniques and their limitations

Traditional replication techniques try to maintain single-copy consistency — they give users an illusion of having a single, highly available copy of data [Bernstein and Goodman 1983; Bernstein et al. 1987].This goal can be achieved in many ways, but the basic concept remains the same: traditional techniques block access to a replica unless it is provably up to date. We call these techniques "pessimistic" for this reason. For example, primary-copy algorithms, used widely in commercial systems, elect a primary replica that is responsible for handling all accesses to a particular object [Bernstein et al. 1987; Dietterich 1994; Oracle 1996]. After an update, the primary synchronously writes the change to the secondary replicas. If the primary crashes, secondaries confer to elect a new primary. Such pessimistic techniques perform well in local-area networks, in which latencies are small and failures uncommon. Given the continuing progress of Internet technologies, it is tempting to apply pessimistic algorithms to wide-area data replication. We cannot expect good performance and availability in this environment, however, for three key reasons.

First, the Internet remains slow and unreliable. The Internet's communication end-to-end latency and availability do not seem to be improving [Zhang et al. 2000; Chandra et al. 2001]. In addition, mobile computers with intermittent connectivity are becoming increasingly popular. A pessimistic replication algorithm, attempting to synchronize with an unavailable site, would block completely. Well-known impossibility results even raise the possiblity that it might corrupt data; for instance it is impossible to agree on a single primary after a failure when network delay is unpredictable [Fischer et al. 1985; Chandra and Toueg 1996].

Second, pessimistic algorithms scale poorly in the wide area. It is difficult to build a large, pessimistically replicated system with frequent updates, because its throughput and availability suffer as the number of sites increases [Yu and Vahdat 2001; Yu and Vahdat 2002]. This is why many Internet and mobile services are optimistic, for instance Usenet [Spencer and Lawrence 1998; Lidl et al. 1994], DNS [Mockapetris 1987; Mockapetris and Dunlap 1988; Albitz and Liu 2001], and mobile file and database systems [Walker et al. 1983; Kistler and Satyanarayanan 1992; Moore 1995; Ratner 1998].

Third, some human activities require asynchronous data sharing. Cooperative engineering or program development often requires people to work in relative isolation. It is better to allow concurrent operations, and to repair occasional conflicts after they happen, than to

lock out the data while someone is editing it.

## 1.2 What is optimistic replication?

Optimistic replication is a group of techniques for sharing data efficiently in wide-area or mobile environments. The key feature that separates optimistic replication algorithms from their pessimistic counterparts is their approach to concurrency control. Pessimistic algorithms synchronously coordinate replicas during accesses and block the other users during an update. In contrast, optimistic algorithms let data be read or written without *a priori* synchronization, based on the "optimistic" assumption that problems will occur only rarely, if at all. Updates are propagated in the background, and occasional conflicts are fixed after they happen. It is not a new idea,[1] but its use has exploded due to the proliferation of the Internet and mobile computing technologies.

Optimistic algorithms offer many advantages over their pessimistic counterparts. First, they improve availability: applications make progress even when network links and sites are unreliable.[2] Second, they are flexible with respect to networking, because techniques such as epidemic replication propagate operations reliably to all replicas, even when the communication graph is unknown and variable. Third, optimistic algorithms should be able to scale to a large number of replicas, because they require little synchronization among sites. Fourth, sites and users are highly autonomous: for example, services such as FTP and Usenet mirroring [Nakagawa 1996; Krasel 2000] let a replica be added with no change to existing sites. Optimistic replication also enables asynchronous collaboration between users, for instance in CVS [Cederqvist et al. 2001; Vesperman 2003] or Lotus Notes [Kawell et al. 1988]. Finally, optimistic algorithms provide quick feedback, as they can apply updates tentatively as soon as they are submitted.

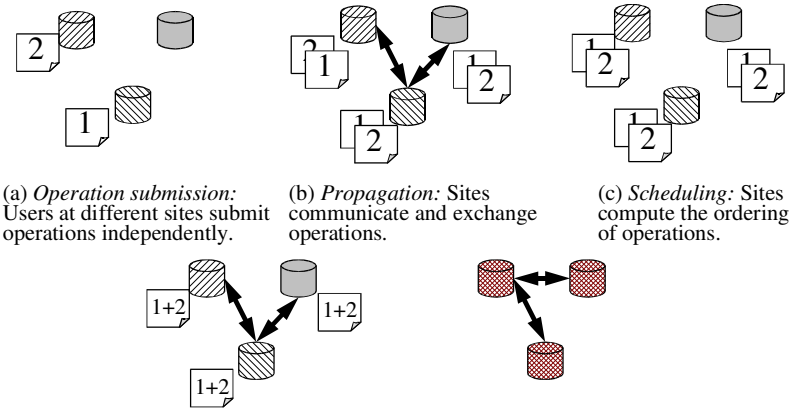
These benefits, however, come at a cost. Any distributed system faces a trade-off between availability and consistency [Fox and Brewer 1999; Yu and Vahdat 2002]. Where a pessimistic algorithm waits, an optimistic one speculates. Optimistic replication faces the unique challenges of diverging replicas and conflicts between concurrent operations. It is thus applicable only for applications that can tolerate occasional conflicts and inconsistent data. Fortunately, in many real-world systems, especially file systems, conflicts are known to be rather rare, thanks to the data partitioning and access arbitration that naturally happen between users [Ousterhout et al. 1985; Baker et al. 1991; Vogels 1999; Wang et al. 2001].

## 1.3 Elements of optimistic replication

This section introduces some basic concepts of optimistic replication and defines common terms used throughout the paper. Figure 1 illustrates how these concepts fit together, and Table 1 provides a reference for common terms. This section provides only a terse overview, as later ones will go into more detail.

1.3.1 *Objects, replicas, and sites.* Any replicated system has a concept of the minimal unit of replication. We call such unit an *object*. A *replica* is a copy of an object stored in a *site*, or a computer. A site may store replicas of multiple objects, but we often use terms replica and site interchangeably, since most optimistic replication algorithms manage each object independently. When describing algorithms, it is useful to distinguish sites that can

[1] Our earliest reference is from Johnson and Thomas [1976], but the idea was certainly developed much earlier.
[2] Tolerating Byzantine (malicious) failures is outside our scope; we cite a few recent papers in this area: Spreitzer et al. [1997], Minsky [2002] and Mazières and Shasha [2002].




(a) *Operation submission:* Users at different sites submit operations independently.

(b) *Propagation:* Sites communicate and exchange operations.

(c) *Scheduling:* Sites compute the ordering of operations.

(d) *Conflict resolution:* Sites detect conflicts and transform offending operations to produce results intended by users.

(e) *Commitment:* Sites agree on the final ordering and reconciliation result. Their changes become permanent.

Fig. 1. Elements of optimistic replication and their roles. Disks represent replicas, memo sheets represent operations, and arrows represent communications between replicas.

update an object — called *master sites* — from those that store read-only replicas. We use the symbol $N$ to denote the total number of replicas and $M$ to denote the number of master replicas for a given object. Common values are $M = 1$ (single-master systems) and $M = N$.

1.3.2 *Operations.* An optimistic replication system must allow access to a replica even while it is disconnected. In this paper, we call a self-contained update to an object an *operation*. To update an object, a user submits an operation at some site. An operation includes a prescription to update the object as well as a *precondition* for detecting conflicts.

The concrete nature of prescriptions and preconditions varies widely among systems. Many systems support only whole-object updates, including Palm [PalmSource 2002] and DNS [Albitz and Liu 2001]. Such systems are called *state-transfer* systems, as they only need to record and transmit the final values of objects, not the sequence of operations. Other systems, called *operation-transfer* systems, allow for more sophisticated descriptions of updates. For example, updates in Bayou [Terry et al. 1995] are written in SQL.

A site applies an operation locally immediately, and it exchanges and applies remote operations in the background. Such systems are said to offer *eventual consistency*, because they guarantee that the state of replicas will converge only eventually. Such a weak guarantee is enough for many optimistic replication applications, but some systems provide stronger guarantees, e.g., that a replica's state is never more than 1 hour old.

1.3.3 *Propagation.* An operation submitted by the user of a replica is tentatively applied to the local replica to let the user continue working based on that update. It is also *logged*, i.e., remembered in order to be propagated to other sites later. These systems often deploy *epidemic propagation* to let all sites receive operations, even when they cannot communicate with each other directly [Demers et al. 1987]. Epidemic propagation lets any two sites that happen to communicate exchange their local operations as well as operations they received from a third site — an operation spreads like a virus does among humans.

1.3.4 *Tentative execution and scheduling.* Because of background propagation, operations are not always received in the same order at all sites. Each site must reconstruct an appropriate ordering that produces an equivalent result across sites and matches the users' intuitive expectations. Thus, an operation is initially considered *tentative*. A site might reorder or transform operations repeatedly until it agrees with others on the final operation ordering. We use the term *scheduling* to refer to the (often non-deterministic) ordering policy.

1.3.5 *Detecting and resolving conflicts.* With no *a priori* site coordination, multiple users may update the same object at the same time. One could simply ignore such a situation — for instance, a room-booking system could handle two requests to the same room by picking one arbitrarily and discarding the other. However, simply dropping concurrent requests is not desirable in many applications, including room booking. This problem is called *lost updates*.

A better way to handle this problem is to detect operations that are in conflict and resolve them, for example, by letting people renegotiate their schedule. A conflict happens when the precondition of an operation is violated, if it is to be executed according to the system's scheduling policy. In many systems, preconditions are built implicitly into the replication algorithm. The simplest example is when all concurrent operations are flagged to be in conflict, as with the Palm Pilot [PalmSource 2002] and the Coda mobile file system [Kumar and Satyanarayanan 1995]. Other systems let users write preconditions explicitly — for example, in a room booking system written in Bayou, a precondition might check the status of the room and disallow double booking [Terry et al. 1995].

Conflict resolution is usually highly application specific. Most systems simply flag a conflict and let users fix it manually. Some systems can resolve a conflict automatically. For example, in Coda, concurrent writes to a '*.o' file can be resolved simply by recompiling the source file [Kumar and Satyanarayanan 1995]. We discuss conflict detection and resolution in more detail in Sections 5 and 6.

1.3.6 *Commitment.* Scheduling and conflict resolution often both involve non-deterministic choices, e.g., regarding ordering of concurrent operations. Moreover, a replica may not have received all the operations that others have. *Commitment* refers to an algorithm to converge the state of replicas by letting sites agree on the set of operations and their final ordering and conflict-resolution results.

## 1.4 Comparison with advanced transaction models

Optimistic replication is related to relaxed (or advanced) transaction models [Elmagarmid 1992; Ramamritham and Chrysanthis 1996]. Both relax the ACID requirements of traditional databases to improve performance and availability, but the motives are different.[3]

Advanced transaction models try to increase the system's throughput by, for example, letting transactions read values produced by non-committed transactions [Pu et al. 1995]. Designed for a single-node or well-connected distributed database, they require frequent communication during transaction execution.

Optimistic replication systems, in contrast, are designed to work with a high degree of

[3]ACID demands that a group of operations, called a transaction, be: Atomic (all-or-nothing), Consistent (safe when executed sequentially), Isolated (intermediate state is not observable) and Durable (the final state is persistent) [Gray and Reuter 1993].

| Term | Meaning | Sections |
|---|---|---|
| Abort | Permanently reject the application of an operation (e.g., to resolve a conflict). | 5.1, 5.5 |
| Clock | A counter used to order operations, possibly (but not always) related to real time. | 4.1 |
| Commit | Irreversibly apply an operation. | 5.1, 5.5 |
| Conflict | Violating the precondition of an operation. | 1.3.5, 3.4, 5, 6 |
| Consistency | The property that the state of replicas stay close together. | 5.1, 5 |
| Divergence control | Techniques for limiting the divergence of the state of replicas. | 8 |
| Eventual consistency | Property by which the state of replicas converge toward one another's. | 5.1 |
| Epidemic propagation | Propagation mode that allows any pair of sites to exchange any operation. | 3.5 |
| Log | A record of recent operations kept at each site. | 1.3.3 |
| Master $(M)$ | A site capable of performing an update locally ($M$ = number of masters). | 1.3.1, 3.1 |
| Object | Any piece of data being shared. | 1.3.1 |
| Operation | Description of an update to the object. | 1.3.2 |
| Precondition | Predicate defining the input domain of an operation. | 1.3.2 |
| Propagate | Transfer an operation to all sites. | 7 |
| Replica $(x_i)$ | A copy of an object stored at a site ($x_i$: replica of object $x$ at site $i$). | 1.3.1 |
| Resolver | An application-provided procedure for resolving conflicts. | 5.4 |
| Schedule | An ordered set of operations to execute. | 3.3, 5.2 |
| Site $(i, j, \ldots, N)$ | A network node that stores replicas of objects ($i, j$: site names; $N$ = number of sites). | 1.3.1 |
| State transfer | Technique that propagates recent operations by sending the object value. | 3.2, 6 |
| Submit | To enter an operation into the system, subject to tentative execution, rollback, reordering, commitment or abort. | 1.3.2 |
| Tentative | Operation applied on isolated replica; may be reordered or aborted. | 1.3.3, 5.5 |
| Timestamp | (See Clock) | |
| Version vector (VV) | (See Vector clock) | |
| Thomas's write rule | "Last-writer wins" algorithm for resolving concurrent updates. | 6.1 |
| Vector clock (VC) | Data structure for tracking order of operations and detecting concurrency. | 4.3 |

Table 1. Glossary of recurring terms.

asynchrony and autonomy. Sites exchange operations in the background and still agree on a common state. They must learn about relationships between operations, often long after they were submitted, and at sites different from where submitted. Their techniques, such as the use of operations, scheduling, and conflict detection, reflect the characteristics of environments for which they are designed. Preconditions play a role similar to traditional concurrency control mechanisms, such as two-phase locking or optimistic concurrency control [Bernstein et al. 1987], but it operates without inter-site coordination. Conflict resolution corresponds to transaction abortion, in that both are designed to fix problems in concurrency control.

That said, there are many commonalities between optimistic replication and advanced transaction models. Epsilon serializability allows transactions to see inconsistent data up to some application-defined degree [Ramamritham and Pu 1995]. This idea has been incorporated into optimistic replication systems; see for example, TACT and session guarantees (Section 8). For another example, Coda's isolation-only transactions apply optimistic concurrency control to a mobile file system [Lu and Satyanarayanan 1995]. It tries to run a set of accesses atomically, but it merely reports an error when atomicity is violated.

### 1.5 Outline

Section 2 overviews several popular optimistic-replication systems and sketches a variety of mechanisms they deploy to manage replicas. Section 3 introduces six key design choices for optimistic replication systems, including the number of masters, state- vs operation transfer, scheduling, conflict management, operation propagation, and consistency guaratees. The subsequent sections examine these choices in more detail.

Section 4 reviews the classic concepts of concurrency and happens-before relationships, which are used pervasively in optimistic replication for scheduling and conflict detection. It also introduces basic techniques used to implement these concepts, including logical and vector clocks. Section 5 introduces techniques for maintaining replica consistency, including scheduling, conflict management, and commitment. Section 6 focuses on a simple subclass of optimistic replication systems, called state-transfer systems, and several interesting techniques available to them. Section 7 focuses on techniques for efficient operation propagation. We examine systems that bound replica divergence in Section 8. Finally, Section 9 concludes by summarizing the systems and algorithms discussed in the paper and offering hints for designers and users of optimistic replication.

The reader may refer to the glossary of recurring notions and terms in Table 1. Also, tables in the conclusion section (Section 9) summarize optimistic replication systems and algorithms along a number of angles.

## 2. APPLICATIONS OF OPTIMISTIC REPLICATION

Optimistic replication is deployed in several major application areas, including wide-area data management, mobile information systems, and computer-based collaboration. This section overviews popular optimistic services to provide a context for the technical discussion that follows.

### 2.1 An internet service: DNS

Optimistic replication is particularly attractive for wide-area network applications, which must tolerate slow and unreliable communication between sites. Examples include caching, and naming or directory services. See for instance WWW caching [Fielding et al. 1999; Chankhunthod et al. 1996; Wessels and Claffy 1997], FTP mirroring [Nakagawa 1996] and directory services such as Grapevine [Birrell et al. 1982], Clearinghouse [Demers et al. 1987], DNS [Mockapetris 1987; Mockapetris and Dunlap 1988; Albitz and Liu 2001], and Active Directory [Microsoft 2000].

DNS (the Domain Name System) is the standard hierarchical name service for the Internet. Names for a particular zone (a sub-tree in the name space) are managed by a single master server that maintains the authoritative database for that zone, and optional slave servers that copy the database from the master. The master and slaves can both answer queries from remote clients and servers. Updating the database takes place on the master, and increments its timestamp. A slave server occasionally polls the master and downloads the database when its timestamp changes.[4] The contents of a slave may lag behind the master's and clients may observe old values.

DNS is a single-master system (all writes for a zone originate at that zone's master) with state transfer (servers exchange the whole database contents). We will discuss these

[4]Recent DNS servers also support proactive update notification from the master and incremental zone transfer [Albitz and Liu 2001].

classification criteria further in Section 3,

### 2.2 Wide-area information exchange: Usenet

Our next example targets a more interactive information exchange. Usenet, a wide-area bulletin board system deployed in 1979, is one of the oldest and still a widely popular optimistically replicated service [Kantor and Rapsey 1986; Lidl et al. 1994; Spencer and Lawrence 1998; Saito et al. 1998]. Usenet originally ran over UUCP, a network designed for intermittent connection over dial-up modem lines [Ravin et al. 1996]. A UUCP site could only copy files to its direct neighbors.

Today's Usenet consists of thousands of sites forming a connected (but not complete) graph built through a series of human negotiations. Each site replicates all news articles,[5] so that a user can read any article from the nearest site. Usenet lets any user post articles to any site. From time to time, articles posted on a site are pushed to the neighboring sites. A receiving site also stores and forwards the articles to its own neighbors. This way, each article "floods" its way through inter-site links eventually to all the sites. Infinite propagation loops are avoided by each site accepting only those articles missing from its disks. An article is deleted from a site by time-out, or by an explicit cancellation request, which propagates among sites just like an ordinary article. Usenet's delivery latency is highly variable, sometimes as long as a week. While users sometimes find it confusing, it is a reasonable cost to pay for Usenet's excellent availability.

Usenet is a multi-master system (an update can originate at any site), that propagates article posting and cancellation operations epidemically.

### 2.3 Personal digital assistants

Optimistic replication is especially suited to environments where computers are frequently disconnected. Mobile data systems use optimistic replication, as in Lotus Notes [Kawell et al. 1988], Palm [Rhodes and McKeehan 1998; PalmSource 2002], Coda [Kistler and Satyanarayanan 1992; Mummert et al. 1995], and Roam [Ratner 1998].

A personal digital assistant (PDA) is a small handheld computer that keeps a user's schedule, address book, and other personal information in a database. Occasionally, the user synchronizes the PDA with his immobile computer or PC. Changes made on the PDA are sent to the PC, and vice-versa. Thus, conflicts can happen, say, when the phone number of a person is changed on both ends. PDAs such as Palm use a simple "modified bits" scheme, taking advantage of the fact that synchronization almost always happens between two particular computers [Rhodes and McKeehan 1998; PalmSource 2002]. Every database item in Palm is associated with a "modified" bit, which is set when the item is updated and reset after synchronization. During synchronization, if only one of the replicas is found to be modified, the new value is copied to the other side. If both the modified bits are set, the system detects a conflict. Conflicts are resolved either by an application-specific resolver or manually by the user.

PDAs represent an example of multi-master, state-transfer systems; a database item is the unit of replication, update, and reconciliation.

[5] In practice, articles are grouped into newsgroups, and a site usually stores only a subset of newsgroups to conserve network bandwidth and storage space. Still, articles posted to a specific newsgroup are replicated on all sites that subscribe to the newsgroup.

### 2.4 A mobile database system: Bayou

Bayou is a research mobile database system [Terry et al. 1995; Petersen et al. 1997]. Bayou lets a user replicate a database on a mobile computer, modify it while being disconnected, and synchronize with any other replica that the user happens to find. Bayou is a complex system because of the challenges of sharing data flexibly in a mobile environment. A user submits updates as high-level operations (SQL statements), which are propagated to other sites epidemically.

A site applies operations tentatively as they are received from the user or from other sites. Because sites may receive operations in different orders, they must undo and redo operations repeatedly as they gradually learn the final order. Conflicts are detected by an explicit precondition (called a dependency check) attached to each operation, and they are resolved by an application-defined merge procedure, also attached to each operation. The final decision regarding ordering and conflict resolution is made by a designated "home," or primary, site. The home site orders operations and resolve conflicts as they arrive and sends the decisions to other sites epidemically as a side effect of ordinary operation propagation.

Bayou is a multi-master, operation-transfer system that uses epidemic propagation over arbitrary, changing communication topologies.

### 2.5 Software version control: CVS

CVS (Concurrent Versions System) is a version control system that lets users edit a group of files collaboratively and retrieve old versions on demand [Cederqvist et al. 2001; Vesperman 2003]. Communication in CVS is centralized through a single site. The central server manages a so-called repository that contains the authoritative copy of the files, along with all changes committed to them in the past. A user creates private copies (replicas) of the files and edits them using standard tools. Any number of users can modify their private copies concurrently. After the work is done, the user commits the private copy to the repository. A commit succeeds immediately if no other user has committed a change to the same files in the interim. If another user has modified a same file but the changes do not overlap, CVS merges them automatically and completes the commit.[6] Otherwise, the user is informed of a conflict, which he or she must resolve manually and re-commit.

CVS is a significant departure from the previous generation of version control tools, such as RCS and SCCS, that pessimistically lock the repository while a user edits a file [Bolinger and Bronson 1995]. CVS supports a more flexible style of collaboration, at the cost of occasional manual conflict resolutions. Most users readily accept this trade-off.

CVS is a multi-master operation-transfer system that centralizes communication through a single repository in a star topology.

### 2.6 Summary

The following table summarizes some of the characteristics of the systems just mentioned. The upcoming sections will detail our classification criteria.

[6] Of course, the updates might still conflict semantically.

| System | # Masters | Operations | Object | Conflict resolution |
|---|---|---|---|---|
| DNS | Single | Update | Database | None |
| Usenet | Multi | Post, cancel | Article | None |
| Palm | Multi | Update | Record | Manual or application-specific |
| Bayou | Multi | SQL | App-defined | Application-specific |
| CVS | Multi | Delete, edit, insert | Line of text | Manual |

## 3. OPTIMISTIC REPLICATION: DESIGN CHOICES

The ultimate goal of any optimistic replication system is to maintain *consistency*; that is, to keep replicas sufficiently similar to one another despite operations being submitted independently at different sites, What exactly is meant by this differs considerably among systems, however. This section overviews how different systems define and implement consistency. We classify optimistic replication systems along the following axes:

| Choice | Description | Effects |
|---|---|---|
| Number of masters | Which replicas can submit updates? | Defines the system's basic complexity, availability and efficiency. |
| Operation definition | What kinds of operations are supported, and to what degree is a system aware of operation semantics? | |
| Scheduling | How does a system order operations? | Defines the system's ability to handle concurrent operations. |
| Conflict management | How does a system define and handle conflicts? | |
| Operation propagation strategy | How are operations exchanged between sites? | Defines networking efficency and the speed of replica convergence |
| Consistency guarantees | What does a system guarantee about the divergence of replica state? | Defines the transient quality of replica state. |

### 3.1 Number of writers: single-master vs. multi-master

This choice determines where an update can be submitted and how it is propagated (Figure 2). *Single-master* systems designate one replica as the master (i.e., $M = 1$). All updates originate at the master and then are propagated to other replicas, or *slaves*. They may also be called *caching* systems. They are simple but have limited availability, especially when the system is write-intensive.

Multi-master systems let updates be submitted at multiple replicas independently and exchange them in the background (i.e., $M > 1$). They are more available but significantly more complex. In particular, operation scheduling and conflict management are issues unique to these systems.

Another potential problem with multi-master systems is their limited scalability due to increased conflict rate. According to Gray et al. [1996], a naïve multi-master system would encounter concurrent updates at the rate of $O(M^2)$, assuming that each master submits operations at a constant rate. The system will treat many of these updates as conflicts and

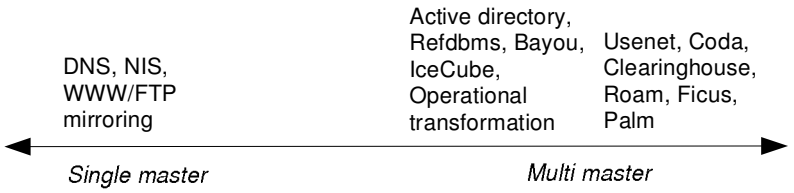



Fig. 2. Single vs. multi-master

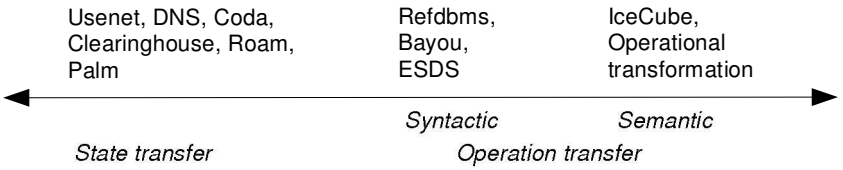



Fig. 3. Definition of operations

resolve them. On the other hand, pessimistic or single-master systems with the same aggregate update rate would experience an abort rate of only $O(M)$, as most of the concurrent operations can be serialized without aborting, using local synchronization techniques, such as two-phase locking [Bernstein et al. 1987]. Still, there are remedies to this scaling problem, as we discuss in Section 7.

### 3.2 Definition of operations: state transfer vs. operation transfer

Figure 3 illustrates the main design choices regarding the definitions of operations. *State-transfer* systems limit an operation either to read or to overwrite an entire object (for some definition of object). *Operation transfer* describes operations more semantically. For example, a state-transfer file system might transfer the entire file (or directory) contents every time a byte is modified, whereas an operation-transfer file system might transfer an operation that produces the desired effect on the file system, sometimes as high-level as "`cc foo.c`" [Lee et al. 2002]. A state-transfer system can be seen as a degenerate form of operation transfer, but there are some qualitative differences between the two types of systems.

State transfer is simple, because maintaining consistency only involves sending the newest replica contents to other replicas. Operation-transfer systems must maintain either a history of operations and have replicas agree on the set of applied operations and their order. On the other hand, they can be more efficient, especially when objects are large and operations are high level; Lee et al. [2002] report a reduction of network traffic by a factor of a few hundreds. Systems that send version deltas, like CVS, are intermediate between state and operation transfer.

Operation transfer also allow for more flexible conflict resolution. For example, in a bibliography database, updates that modify the authors of two different books can both be accommodated in operation-transfer systems (semantically, they do not conflict), but it is difficult to do the same when a system transfers the entire database contents every time [Golding 1992; Terry et al. 1995].

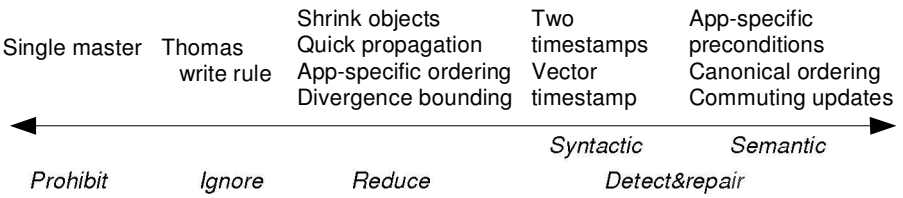



Fig. 4. Design choices regarding conflict handling.

### 3.3 Scheduling: syntactic vs. semantic

The goal of scheduling is to order operations in a way expected by users, and to make replicas produce equivalent states. Scheduling policies can be classified into *syntactic* and *semantic* policies (Figure 3). Syntactic scheduling sorts operations based only on information about when, where and by whom operations were submitted.

Syntactic methods are simple but may cause unnecessary conflicts. Consider, for example, a system for reserving some equipment on loan, where the pool initially contains a single item. Three requests are submitted concurrently: (1) User A requests an item, (2) User B requests an item, and (3) User C adds an item to the pool. If a site schedules the requests syntactically in the order 1, 2, 3, then request 2 will fail (B cannot borrow from an empty pool). If the system is aware of the operation semantics, it could order 1, 3, then 2, thus satisfying all the requests.

Syntactic scheduling is simple and generic. The most popular example is ordering operations by timestamp. Another syntactic example is giving priority to a manager's operations over his employees'. However, as the above example shows, it may bring unnecessary conflicts. As there is no single total order of operations in a distributed system, syntactic mechanisms may involve rescheduling.

Semantic scheduling, on the other hand, exploits semantic properties such as commutativity of idempotency of operations. This can avoid conflicts or reduce the amount of roll-back when a site computes a new schedule. In a replicated file system, for instance, writing to two different files commutes, as does creating two different files in the same directory. The file system may order such pairs of operations in any way and replicas still converge [Balasubramaniam and Pierce 1998; Ramsey and Csirmaz 2001]. Semantic scheduling is seen only in operation-transfer systems, since state-transfer ones systems are oblivious to operations. Semantic scheduling increases scheduling flexibility and reduces conflict, but at the cost of application dependence and complexity.

We will discuss the techniques for determining ordering in more detail in Sections 4 and 5.

### 3.4 Handling conflicts

Conflicts happen when some operations fail to satisfy their preconditions. Figure 4 presents a taxonomy of approaches for dealing with conflicts.

The best approach is to prevent conflicts from happening altogether. Pessimistic algorithms prevent conflicts by blocking or aborting operation as necessary. Single-master systems avoid conflicts by accepting updates only at one site (but allow reads to happen anywhere). These approaches, however, come at the cost of lower availability, as discussed in Section 1. Conflicts can also be reduced, for example, by quickening propagation or by dividing objects into smaller independent units.

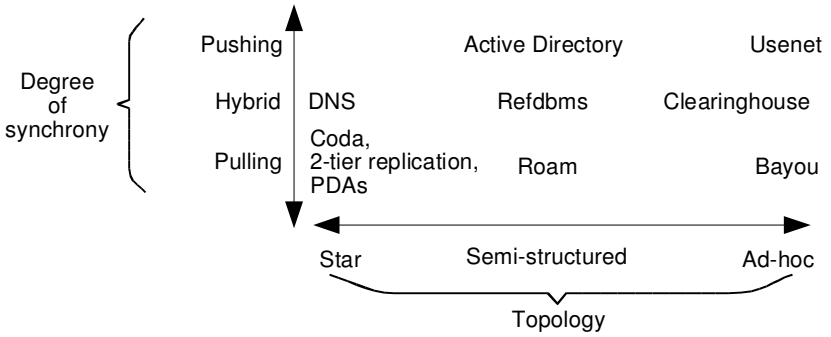



Fig. 5. Design choices regarding operation propagation.

Some systems ignore conflicts: any potentially conflicting operation is simply overwritten by a newer operation. Such *lost updates* may not be an issue if the loss rate is negligible, or if users can voluntarily avoid lost updates. A distributed name service is an example, where only the owner of a name may modify it, so avoiding lost updates is easy [Demers et al. 1987; Microsoft 2000].

The user experience is improved when a system can detect and signal conflicts, as discussed in Section 1.3.5. Conflict detection policies are also classified into *syntactic* and *semantic* policies. In systems with syntactic conflict detection policies, preconditions are not explicitly specified by the user or the application. They rely only on the timing of operation submission and conservatively declare a conflict between any two concurrent operations. Section 4 introduces various techniques for detecting concurrent operations. Systems with semantic knowledge of operations can often exploit that to reduce conflicts. For instance, in a room-booking application, two concurrent reservation requests to the same room object could be granted, as long as their duration does not overlap.

The trade-off between syntactic and semantic conflict detection parallels that of scheduling: syntactic policies are simpler and generic but cause more conflicts, whereas semantic policies are more flexible, but application specific. In fact, conflict detection and scheduling are closely related issues: syntactic scheduling tries to preserve the order of non-concurrent operations, whereas syntactic conflict detection flags any operations that are concurrent. Semantic policies are attempts to better handle such concurrent operations.

## 3.5 Propagation strategies and topologies

Local operations must be transmitted and re-executed at remote sites. Each site will record its changes (called logging) while disconnected from others, decide when to communicate with others, and exchange changes with other sites. Propagation policies can be classified along two axes, communication topology and the degree of synchrony, as illustrated in Figure 5.

Fixed topologies, such as a star or spanning tree can be very efficient, but work poorly in dynamic, failure-prone network environments. At the other end of the spectrum, many optimistic replication systems rely on *epidemic communication* that allows operations to propagate through any connectivity graph even if it changes dynamically [Demers et al. 1987].

The degree of synchrony shows the speed and frequency by which sites communicate

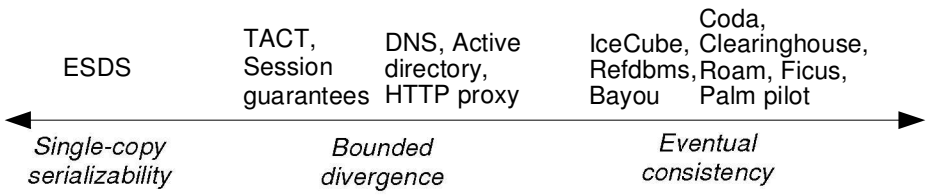



Fig. 6. Choices regarding consistency guarantees

and exchange operations. At one end of the spectrum, pull-based systems demand that each site poll other sites either manually (e.g., PDAs) or periodically (e.g., DNS) for new operations. In push-based systems, a site with new updates proactively sends them to others. In general, the quicker the propagation, the less the degree of replica inconsistency and the rate of conflict, but more the complexity and overhead, especially when the application receives many updates relative to read requests.

### 3.6 Consistency guarantees

In any optimistic replication system, the states of replicas may diverge somewhat. A consistency guarantee specifies whether a client application may observe divergence, and how much. Figure 6 shows some common choices.

Single-copy serializability (1SR) ensures that a set of all accesses by all sites produces an effect that is equivalent to some serial execution of them at a single site [Bernstein and Goodman 1983].

At the other end of the spectrum, *eventual consistency* guarantees only that the state of replicas will eventually converge. In the meantime, applications may observe arbitrarily stale state, or even incorrect state. We define eventual consistency a bit more formally in Section 5.1. Eventual consistency is a fairly weak concept, but it is the guarantee offered by most optimistic-replication systems, for which the availability is of paramount importance. As such, most of the techniques we describe in this paper are for maintaining eventual consistency.

In between single-copy and eventual consistencies, numerous intermediate consistency types have been proposed, which we call "bounded divergence" [Ramamritham and Chrysanthis 1996; Yu and Vahdat 2001]. Bounded divergence is usually achieved by blocking accesses to a replica when certain consistency conditions are not met. Techniques for bounding divergence are covered in Section 8.

## 4. DETECTING CONCURRENCY AND HAPPENS-BEFORE RELATIONSHIPS

An optimistic replication system accepts independently submitted operations, then orders them and (often) detects conflicts. Many systems use some intuitive relations between operations as the basis for this task. This section reviews these relations and techniques for expressing them.

### 4.1 The happens-before and concurrency relations

Scheduling requires a system to know which events happened in which order. However, in a distributed environment in which communication delays are unpredictable, we cannot define a natural total ordering between events. The concept of *happens-before* is an implementable partial ordering that intuitively captures the relations between distributed events

```
var vc: array [1..M] of Timestamp
proc SubmitOperation(op) // Called when this site submits a new operation op
    vc[myself] := vc[myself] + 1
    op.issuer := myself
    op.vc := vc
    ... send op to other sites ...
proc ReceiveUpdate(op) // Called when an operation op arrives from a remote site
    // Here, we assume that operations from a single site arrives in FIFO order
    vc[op.issuer] = op.vc[op.issuer]
    ... apply the operation ...
```

Fig. 7. Generating vector clocks. Every site executes the same algorithm. Variable *myself* is the name of the current site.

[Lamport 1978]. Consider two operations α and β submitted at sites $i$ and $j$, respectively. Operation α *happens before* β when:

— $i = j$ and α was submitted before β, or

— $i \neq j$ and β is submitted after $j$ has executed α,[7] or

— There exists an operation γ, such that α happens before γ and γ happens before β.

If neither operation α or β happens before the other, they are said to be *concurrent*.

The happens-before and concurrency relations are used in a variety of ways in optimistic replication, e.g., as a hint for operation ordering (Section 5.2), to detect conflicts (Section 5.3), and to propagate operations (Section 7.1). The following sections review algorithms for representing or detecting happen-before and concurrency relations.

## 4.2 Explicit representation

Some systems represent the happens-before relation simply by attaching, to an operation, the names of the operations that precede it [Birman and Joseph 1987; Mishra et al. 1989; Fekete et al. 1999; Kermarrec et al. 2001; Kang et al. 2003]. Operation α happens-before β if α appears in β's predecessors. The size of this set is independent of the number of replicas, but it grows with the number of past operations.

## 4.3 Vector clocks

A *vector clock* (VC), also called a version vector, timestamp vector, or a multi-part timestamp, is a compact data structure that accurately captures the happens-before relationship [Parker et al. 1983; Fidge 1988; Mattern 1989]. VCs are proved to be the smallest such data structure by Charron-Bost [1991].

A vector clock $VC_i$, kept on Site $i$, is an $M$-element array of timestamps.[8] A timestamp is any number that increases for every distinct event — it is commonly just an integer counter. To submit a new operation α, Site $i$ increments $VC_i[i]$ and attaches the new value of $VC_i$, now called α's timestamp $VC_\alpha$, to α. The current value of $VC_i[i]$ is called $i$'s timestamp, as it shows the last time an operation was submitted at Site $i$. If $VC_i[j] = T$, this means that Site $i$ has received all the operations with timestamps up to $T$, submitted at Site $j$. Figure 7 shows how VCs are computed.

[7] Site $j$ must have previously received α from Site $i$.

[8] $M$ denotes the number of master replicas (Section 1.3.1). In practice, vector clocks are usually implemented as a table that maps the site's name (say, IP address) to a timestamp.

```
var clock: Timestamp // Logical clock
proc SubmitOperation(op) // Called when this site submits a new operation op.
    clock := clock + 1
    op.clock := clock
    ... send op to other sites ...
proc ReceiveUpdate(op) // Called when an operation op arrives from a remote site.
    clock := max(clock, op.clock) + 1
    ... apply the operation ...
```

Fig. 8. Generating logical clocks. Every site executes the same algorithm.

$VC_\beta$ *dominates* $VC_\alpha$ if $VC_\alpha \neq VC_\beta$ and $\forall k \in \{1 \ldots M\}$, $VC_\alpha[k] \leq VC_\beta[k]$. Operation $\alpha$ happens before $\beta$ if and only if $VC_\beta$ dominates $VC_\alpha$. If neither VC dominates the other, the operations are concurrent.

A general problem with VCs is size when $M$ is large, and complexity when sites come and go dynamically, although solutions exist [Ratner et al. 1997; Petersen et al. 1997; Adya and Liskov 1997].

## 4.4 Logical and real-time clocks

A single, scalar timestamp can be also used to express happens-before relationships. This section reviews several types of scalar timestamps and their characteristics.

A logical clock, also called a Lamport clock, is a timestamp maintained at each site, as illustrated in Figure 8 [Lamport 1978]. When submitting an operation $\alpha$, the site increments the clock and attaches the new value, noted $C_\alpha$, to $\alpha$. Upon receiving operation $\alpha$, the receiver sets its logical clock to be a value larger than either its current value or $C_\alpha$. With this definition, if an operation $\alpha$ happens before $\beta$, then $C_\alpha < C_\beta$. However, logical clocks (and any scalar clocks) cannot detect the concurrency, because $C_\alpha < C_\beta$ does *not* necessarily imply that $\alpha$ happens before $\beta$.

Real-time clocks (RTC) can also be used to track happens-before. Comparing RTCs between sites, however, is meaningful only if they are properly synchronized. Consider two operations $\alpha$ and $\beta$, submitted at sites $i$ and $j$, respectively. Even if $\beta$ is submitted after $j$ received $\alpha$, $\beta$'s timestamp could still be smaller than $\alpha$'s if $j$'s clock lags far behind $i$'s. This situation cannot ultimately be avoided, because clock synchronization is a best-effort service in asynchronous networks [Chandra and Toueg 1996]. Modern algorithms such as NTP, however, can keep clock skew within tens of microseconds in a LAN, and tens of milliseconds in a wide area with a negligible cost [Mills 1994]. This is enough to capture most happens-before relations that happen in practice.

Real-time clocks do have an advantage over logical and vector clocks: they can capture relations that happen via a "hidden channel", or outside the system control. Suppose that a user submits an operation $\alpha$ on computer $i$, walks over to another computer $j$, and submits annother operation $\beta$. For the user, $\alpha$ clearly happens before $\beta$, and real-time clocks can detect that. Other clocks may not detect such a relation, because $i$ and $j$ might never have exchanged messages before $\beta$ was submitted.

## 4.5 Plausible clocks

Plausible clocks combine ideas from logical and vector clocks to build clocks with intermediate strength [de Torres-Rojas and Ahamad 1996]. They have the same theoretical strength as scalar clocks, but better practical accuracy. The paper introduces a variety of plausible clocks, including the use of a vector clock of fixed size $K$ ($K \leq M$), with Site $i$

using $(i \bmod K)$th entry of the vector. This vector clock can often (but not always) detect concurrency.

## 5. CONCURRENCY CONTROL AND EVENTUAL CONSISTENCY

A site in an optimistic replication system collects and orders operations submitted independently at this and other sites. This section reviews techniques for achieving an eventual consistency of replicas in such environments. We first define eventual consistency using the concepts of schedule and its equivalence. We subsequently examine the necessary steps toward this goal: computing an ordering, identifying and resolving conflicts, and commitment.

### 5.1 Eventual consistency

Informally, eventual consistency means that replicas eventually reach the same final value, if users stop submitting new operations. This section tries to clarify this concept, especially when in practice sites independently submit operations continually.

We define two schedules to be *equivalent* when, starting from the same initial state, they produce the same final state. Schedule equivalence is an application-specific concept; for instance, if a schedule contains consecutive commuting operations, swapping their order preserves the equivalence. For the purpose of conflict resolution, we also allow some operation $\alpha$ to be included in a schedule, but not executed. We use the symbol $\overline{\alpha}$ to denote such a voided operation.

*Definition:* A replicated object is eventually consistent when it meets the following conditions, assuming that all replicas start from the same initial state.

—At any moment, for each replica, there is a prefix of the schedule that is equivalent to a prefix of the schedule of every other replica. We call this a *committed* prefix for the replica.

—The committed prefix of each replica grows monotonically over time.

—All non-voided operations in the committed prefix satisfy their preconditions.

—For every submitted operation $\alpha$, either $\alpha$ or $\overline{\alpha}$ will eventually be included in the committed prefix.

This general definition leaves plenty of room for differing implementations. The basic trick is to play with equivalence and with preconditions to allow for more scheduling flexibility. For instance, in Usenet, the precondition is always true, it never voids an operation, and thus it applies postings in any order; eventual consistency reduces to eventual delivery of operations. In Bayou, in contrast, allows explicit preconditions to be written by users or applications, and requires that operations applied in the same order at every site.

### 5.2 Scheduling

As introduced in Section 3.3, scheduling policies in optimistic replication systems vary along the spectrum between syntactic and semantic approaches. Syntactic scheduling defines a total order of operations from the timing and location of operation submission, whereas semantic approaches provide more scheduling freedom by exploiting operation semantics.

5.2.1 *Timestamp operation ordering.* A scheduler should at least try to preserve the happens-before relationships seen by operations. Otherwise, users may observe an object's state to "roll back" randomly and permanently, which renders the system practically useless. Timestamp scheduling is a straightforward attempt toward this goal.

A typical timestamp scheduler uses a scalar clock technique to order operations. Examples include Active Directory [Microsoft 2000], Usenet [Spencer and Lawrence 1998], and TSAE [Golding 1992]. In the absence of concurrent updates, vector clocks also provide a total ordering, as used in LOCUS [Parker et al. 1983; Walker et al. 1983], and Coda [Kistler and Satyanarayanan 1992; Kumar and Satyanarayanan 1995]. Systems that maintain an explicit log of operations, such as Bayou, can use an even simpler solution: exchange the log contents sequentially [Petersen et al. 1997]. Here, a newly submitted operation is appended to the site's log. During propagation, a site simply receives missing operations from another site and append them to the log in first-in-first-out order. These systems are effectively using the log position of an operation as a logical clock.

Syntactic policies order concurrent operations in some arbitrary order. In some systems, e.g., those that use scalar timestamps, sites can order concurrent operations deterministically. Other systems, including Bayou, may produce different orderings at different sites. They must be combined with an explicit commitment protocol to let sites eventually agree on one ordering. We will discuss such protocols in Section 5.5.

5.2.2 *Semantic scheduling: Exploiting commutativity.* Semantic scheduling techniques take the semantic relations between operations into account, either in addition to the happens-before relationship, or instead of it. A common example is the use of commutativity [Jagadish et al. 1997]. If two consecutive operations $\alpha$ and $\beta$ commute, they can run in either order, even if related by happens-before. This enables to reduce the number of rollbacks and redos when a tentative schedule is re-evaluated.

A replicated dictionary (or table) is a popular example, where all dictionary operations (insertion and deletion) with different keys commute with each other [Wuu and Bernstein 1984; Mishra et al. 1989].

5.2.3 *Semantic scheduling: Canonical ordering.* Ramsey and Csirmaz [2001] formally study optimistic replication in a file system. For every possible pair of concurrent operations, they define a rule that specifies how they interact and may be ordered (non-concurrent operations are applied in their happens-before order.) For instance, they allow creating two files `/a/b` and `/a/c` in any order, even though they both update the same directory. Or, if one user modifies a file, and another deletes its parent directory, it marks them as conflicting and asks the users to repair them manually. Ramsey and Csirmaz [2001] prove that this algebra in fact keeps a file system consistent and converges the state of replicas.

This file system supports few operation types, including create, remove, and edit. In particular, it lacks "move", which would have increased the complexity significantly, as moving a file involves three objects: two directories and a file. Despite the simplification, the algebra contains 51 different rules. It remains to be seen how this approach applies to more complex environments.

5.2.4 *Operational transformation.* Operational transformation (OT) is a technique developed for collaborative editors. A command by a user, e.g., text insertion or deletion, is applied at the local site immediately, and then sent to other sites. Sites apply remote commands in reception order, and do not reorder already-executed operations; thus two

sites apply the same operations, but possibly in different orders. For every possible pair of concurrent operations, OT defines a rewriting rule to preserve its intention and converge the state of replicas, regardless of reception order. Thus OT uses semantics to schedule efficiently, and transforms operations to run in any order even when they do not naturally commute. Some references are Ellis and Gibbs [1989], Sun and Ellis [1998], Sun et al. [1996], Sun et al. [1998] and Vidot et al. [2000].

Consider a text editor that shares a text "abc". The user at site *i* executes insert("X", 1), yielding "Xabc", and sends the update to Site *j*. The user at site *j* executes delete(1) yielding "bc", and sends the update to Site *i*. In a naïve implementation, Site *j* would have "Xbc", whereas Site *i* would have an unexpected "abc". Using OT, Site *i* rewrites *j*'s operation to delete(2).

The actual set of rewriting rules is complex and non-trivial, because it must provably converge the state of replicas, given arbitrary pairs of concurrent operations [Cormack 1995; Vidot et al. 2000]. The problem becomes even more complex when one wants to support three or more concurrent users [Sun and Ellis 1998]. Palmer and Cormack [1998] prove the correctness of transformations for a shared spreadsheet that supports operations such as updating cell values, adding or deleting rows or columns, and changing formulæ. Molli et al. [2003] extend the OT approach to support a replicated file system.

5.2.5 *Semantic scheduling: Combinatorial-optimization approach.* IceCube is a toolkit that supports multiple applications and data types using a concept called *constraints* between operations [Kermarrec et al. 2001; Preguiça et al. 2003]. Constraints can be supplied from several sources: the user, the application, a data type, or the system.

IceCube supports several kinds of constraints, including dependence ($\alpha$ executes only after $\beta$ does), implication (if $\alpha$ executes, so does $\beta$), choice (either $\alpha$ or $\beta$ may applied, but not both), and a specialized constraint for expressing resource allocation timings [Matheson 2003]. For instance, a user might try to reserve Room 1 or 2 (choice); if Room 2 is chosen, rent a projector (implication), which is possible only if sufficient funds are available (dependence).

IceCube treats scheduling as an optimization problem, where the goal is to find the "best" schedule of operations compatible with the stated constraints. The goodness of a schedule is defined by the user or the application — e.g., one may define a schedule with fewer conflicts to be better. Furthermore, IceCube supports an explicit commutativity relation to subdivide the search space. Despite the NP-hard nature of the problem, IceCube uses an efficient hill-climbing-based constraint solver that can order a benchmark of 10,000 operations in less than 3 seconds [Preguiça et al. 2003].

## 5.3 Detecting conflicts

An operation $\alpha$ is in conflict when its precondition is unsatisfied, given the state of the replica after tentatively applying all operations before $\alpha$ in the current schedule. Conflict management involves two subtasks: detecting a conflict, the topic of this section, and resolving it, which we will review in Section 5.4. Just like for scheduling, techniques range over the spectrum between syntactic and semantic approaches.

Many systems do nothing about conflict, for instance any system using the Thomas's write rule (Section 6.1). These systems simply apply operations in the order of schedule, oblivious of any concurrency that might exist between them. Detecting and explicitly resolving conflicts, however, alleviates the lost-update problem and helps users better man-

age data, as discussed in Section 1.3.5.

Syntactic conflict detection uses the happens-before relationship, or some approximation, to flag conflicts. That is, an operation is deemed in conflict when it is concurrent with another operation. We describe syntactic approaches in more detail in Section 6 in the context of state transfer systems, because that is where they are the most often used.

Semantic approaches use knowledge of operation semantics to detect conflicts. In some systems, the conflict detection procedure is built in. For instance, in a replicated file system, creating two different files concurrently in the same directory is not a conflict, but updating the same regular file concurrently is a conflict [Ramsey and Csirmaz 2001; Kumar and Satyanarayanan 1993]. Other systems, notably Bayou and IceCube, let the application or the user write explicit preconditions. This approach isolates the application-independent components of optimistic replication — e.g., operation propagation and commitment — from conflict detection and resolution. Semantic policies are strictly more expressive than syntactic counterparts, since one can easily write a semantic conflict detector that emulates a syntactic algorithm. For instance, Bayou [Terry et al. 1995] can be programmed to detect conflict using the two-timestamp algorithm presented in Section 6.2.

Most operation-transfer systems use semantic conflict detector, mainly because the application already describes operations semantically — adding an application-specific precondition require little additional engineering effort. On the other hand, state-transfer systems use both approaches.

## 5.4 Resolving conflicts

The role of conflict resolution is to rewrite or abort offending operations to remove suspected conflicts. Conflict resolution can be either manual or automatic. Manual conflict resolution simply excludes the offending operation from the common schedule and presents two versions of the object. It is up to the user to create a new, merged version and re-submit the operation. This strategy is used by systems such as Lotus [Kawell et al. 1988], Palm [PalmSource 2002], and CVS (Section 2.5).

5.4.1 *Automatic conflict resolution in file systems.* Automatic conflict resolution is performed by an application-specific procedure that takes two versions of an object and creates a new one. Such an approach is well studied in replicated file systems, such as LOCUS [Walker et al. 1983], Ficus, Roam [Reiher et al. 1994; Ratner 1998], and Coda [Kumar and Satyanarayanan 1995]. For instance, concurrent updates on a mail folder file can be resolved by computing the union of the messages from the two replicas. Concurrent updates to object (`*.o`) files can be resolved by recompiling from their source.

5.4.2 *Conflict resolution in Bayou.* Bayou supports multiple applications types by attaching an application-specific precondition (called the *dependency check*) and resolver (called the *merge procedure*) to each operation. Every time an operation is added to a schedule or its schedule ordering changes, Bayou runs the dependency check; if it fails, Bayou runs the merge procedure, which can perform any fix-up necessary. For instance, if the operation is an appointment request, the dependency check might discover that the requested slot is not free any more; then the merge procedure could try a different time slot.

To converge the state of replicas, every merge procedure must be completely deterministic, including its failure behavior (e.g., it may not succeed on some site and run out of memory on another). Practical experience with Bayou has shown that it is difficult to write

merge procedures for all but the simplest of cases [Terry et al. 2000].

## 5.5 Commitment protocols

Commitment serves three practical purposes. First, when sites can make non-deterministic choices, during scheduling conflict resolution, commitment ensures sites agree about them. Second, it lets users know which operations are stable, i.e., their effect will never be rolled back. Third, commitment acts as a space-bounding mechanism, because information about stable operations can safely be deleted from the site.

5.5.1 *Commitment by common knowledge.* Many systems can do without explicit commitment. Examples include systems that use totally deterministic scheduling and conflict-handling algorithms, including single-master systems (DNS and NIS) and systems that use Thomas's write rule (Usenet, Active Directory). These systems can rely on timestamps to order operations deterministically and conflicts are either nonexistent or just ignored.

5.5.2 *Agreement in the background.* The mechanisms introduced in this section let sites agree on the set of operations known to be received at all sites. TSAE (Time-Stamped Anti Entropy) is an operation-transfer algorithm that uses real-time clocks to schedule operations syntactically. TSAE uses *ack vectors* in conjunction with vector clocks (Section 7.1) to let each site learn about the progress of other sites. The ack vector $AV_i$ on Site $i$ is an $N$-element array of timestamps. $AV_i[i]$ is defined to be $\min_{j\in\{1\ldots M\}}(VC_i[j])$, i.e., Site $i$ has received all operations with timestamps no newer than $AV_i[i]$, regardless of their origin. Ack vectors are exchanged among sites and updated by taking pair-wise maxima, just like VCs. Thus, if $AV_i[k] = t$, then $i$ knows that $k$ has received all messages up to $t$. Figure 9 illustrates the relationship among operations, the schedule, and ack vectors. With this definition, all operations with timestamps older than $\min_{j\in\{1\ldots N\}}(AV_i[j])$ are guaranteed to have been received by all sites, and they can safely be executed in the timestamp order and deleted. For liveness and efficiency, this algorithm must use loosely synchronized real-time clocks (Section 4.4) for timestamps. Otherwise, a site with a very slow timestamp could stall the progress of ack vectors of all other sites. Moreover, even a single unresponsive site could stall the progress of ack vectors on all other sites. This problem becomes more likely as the number of sites increases.

ESDS is also an operation-transfer system, but it uses non-deterministic syntactic policy to order concurrent operations. Each operation in ESDS is associated with a set of operations that should happen before it, using a graph representation (Section 4.2). For each operation, each site independently assigns a timestamp that is greater than those that happen before it. The final commitment order is defined by the minimal timestamp assigned to each operation. Thus, a site can commit an operation $\alpha$ when it receives $\alpha$'s timestamps from all other sites, and it has committed all operations that happen before $\alpha$.

Neither TSAE nor ESDS performs any conflict detection or resolution. Their commitment protocols are thus simplified — they only need to agree on the set of received operations and their order.

5.5.3 *Commitment by consensus.* Some systems use consensus protocols to agree on which operations to be committed or aborted and in which order [Fischer et al. 1985].

The *primary-based commitment* protocol, used in Bayou, designates a single site as the *primary* that makes such decisions unilaterally [Petersen et al. 1997]. The primary orders operations as they arrive (Section 5.2.1) and commits operations by assigning them mono-

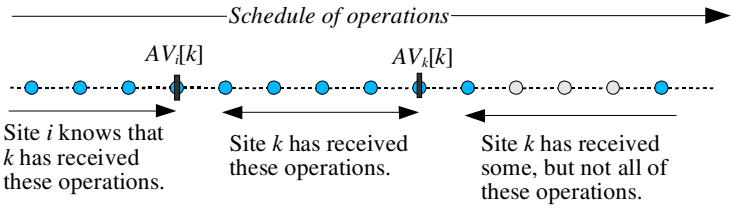



Fig. 9. Relationship between operations, schedule, and ack vectors. The circles represent operations, ordered according to an agreed-upon schedule. $AV_i[k]$ shows a conservative estimate of operations received by $k$. It is no larger than $AV_k[k]$, which itself is a conservative representation of the set of operations that $k$ has received.

tonically increasing *commit sequence numbers* (CSN). The mapping between operations and their CSNs is transmitted as a side effect of ordinary operation propagation process. Other sites commit operations in the CSN order and delete them from the log. Notice the difference between Bayou and single-master systems. In the latter, the lone master submits updates and commits them immediately. Other sites must submit changes via the master. In contrast, Bayou allows any site to submit operations and propagate them epidemically and users to see the effects of operations quickly.

Deno uses a quorum-based commitment protocol [Keleher 1999]. Deno is a pessimistic system that yet exchanges messages epidemically. Deno decides the outcome of each operation independently. A site that wishes to commit an operation runs a two-phase weighted voting [Gifford 1979]. Upon receiving a commit request, a site votes in favor of the update if the operation does not conflict locally with any prior operations. When a site observes that votes for an operation have reached a majority, it locally commits the operation and sends a commit notice to other sites. Simulation results suggest that the performance of this protocol is similar to a classic single-master scheme in the common case when no site has failed. Even though Deno is a pessimistic system, the idea of commitment using weighted voting should apply to optimistic environments as well.

## 5.6 Summary

Eventual consistency involves agreement over the scheduling of operations: while tentative state of replicas might diverge, sites must eventually agree on the contents and ordering of a committed prefix of their schedules. The following table summarizes the techniques discussed in this section for this task.

| Problem | Solution | Advantages | Disadvantages |
|---|---|---|---|
| Ordering | Syntactic | Simple, generic | Unnecessary conflicts |
| | Commuting operations | Simple | App-specific, limited applicability |
| | Canonical ordering | Formal | App-specific, limited applicability |
| | Operational transformation | Formal | Complexity, limited applicability |
| | Semantic optimization | Expressive, powerful | Complexity |
| Conflicts | Syntactic | Simple, generic | Unnecessary conflicts |
| | Semantic | Reduces conflicts, expressive | App-specific |
| Commitment | Common knowledge | Simple | Limited applicability |
| | Ack vector | — | Weak liveness |
| | Consensus | — | Complex |

## 6. STATE-TRANSFER SYSTEMS

State-transfer systems can be considered degenerate instances of operation-transfer systems. Nonetheless, they allow for some interesting techniques — because an operation always overwrites the entire object, replicas can converge simply by transferring the contents of the newest replica to others. Section 6.1 discusses a simple and popular technique called Thomas's write rule. Sections 6.2 to 6.4 introduce several algorithms that enable more refined conflict detection and resolution.

### 6.1 Replica-state convergence using Thomas's write rule

State-transfer systems need to agree only on which replica stores the newest contents. *Thomas's write rule* is the most popular epidemic algorithm for achieving eventual consistency [Johnson and Thomas 1976; Thomas 1979]. Here, each replica stores a timestamp (Section 4.4) that represents the "newness" of its contents. Occasionally, a replica, say $i$, retrieves another replicas $j$'s timestamp. If $j$'s timestamp is newer than $i$'s, $i$ copies $j$'s contents and timestamp to itself. Figure 10 shows the pseudocode of Thomas's write rule. This algorithm does not detect conflicts — it silently discards contents with older timestamps. Systems that need to detect conflicts will use algorithms described later in this section.

With Thomas's write rule, deleting an object requires special treatment. Simply deleting a replica and its associated timestamp could cause an *update/delete ambiguity*. Suppose that Site $i$ updates the object contents (timestamp $T_i$), and Site $j$ deletes the object (timestamp $T_j$) simultaneously. Later, Site $k$ receives the update from $j$ and deletes the replica and timestamp from disk. Site $k$ then contacts Site $i$. The correct action for $k$ would be to create a replica when $T_i > T_j$, and ignore the update otherwise; but because it no longer stores the timestamp, Site $k$ cannot make that decision.

Two solutions have been proposed to address the update/delete ambiguity. The first solution is simply to demand an off-line, human intervention to delete objects, as in DNS [Albitz and Liu 2001] and NIS [Sun Microsystems 1998]. The second solution is to use so-called "death certificates" or "tombstones," which maintain the timestamps (but not the contents) of deleted objects on disk. This idea is used by Fischer and Michael [1982], Clearinghouse [Demers et al. 1987], Usenet [Spencer and Lawrence 1998], and Active Directory [Microsoft 2000].

```
—— Per-object, persistent data structures at each site ——
var state: Data
    ts: Timestamp
—— Called when the site updates the object ——
proc SubmitUpdate(newState)
   state := newState
   ts := CurrentTime()
—— Receiver side: called occasionally ——
proc ReceiveUpdate(src)
   srcTs := Receive src's timestamp.
   if ts < srcTs then
      state := Receive src's state.
      ts := srcTs
```

Fig. 10. State propagation using Thomas's write rule. Each object keeps timestamp *ts* that shows the last time it was updated, and contents *data*. An update is submitted by a site by *SubmitUpdate*. Each site calls *ReceiveUpdate* occasionally and downloads a peer's contents when its own timestamp is older than the peer's.

```
—— Per-object, persistent data structures at each site ——
var state: Data
    ts, prevTs: Timestamp
—— Called when the site updates the object ——
proc SubmitUpdate(newState)
   state := newState
   if this is the first update since the last synchronization then
      prevTs := ts
   ts := CurrentTime()
—— This procedure runs on both sides when two sites exchange their state——
proc Synchronize(src)
   srcTs, srcPrevTs := Receive src's ts and prevTs.
   if prevTs ≠ srcPrevTs then
      A conflict detected; resolve
   elif ts < srcTs then
      // The object is updated only on src
      state := Receive src's state.
      ts := srcTs
```

Fig. 11. Operation propagation and conflict detection using the two-timestamp algorithm. An update is submitted locally by *SubmitUpdate*. Two sites synchronize occasionally, and they both call Synchronize to retrieve the timestamps and data of the peer.

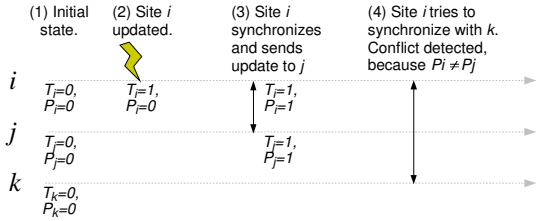



Fig. 12. An example of erroneous conflict detection using the two-timestamp algorithm. A lightening bolt shows the submission of an operation, and an arrow shows bidirectional operation propagation. $T_x$ shows the current timestamp of replica $x$ (noted *ts* in Figure 11), and $P_x$ show its previous timestamp (i.e., *prevTs*). Initially in (1), the contents of the replicas are identical, with $T_x = P_x = 0$ for all the replicas. In step (4), Replicas $i$ and $k$ try to synchronize. The algorithm incorrectly detects a conflict, because $P_i(=2) \neq P_k(=0)$. In reality, Replica $k$ is strictly older than Replica $i$.

### 6.2 Two-timestamp algorithm

The two-timestamp algorithm is an extension to Thomas's write rule to enable conflict detection [Gray et al. 1996; Balasubramaniam and Pierce 1998]. Here, a replica $i$ keeps a timestamp that shows the newness of the data, and a "previous" timestamp that shows the last time the object was updated. A conflict is detected when the previous timestamps from two sites differ. Figure 11 shows the pseudocode. The same logic is sometimes also used by operation-transfer systems to detect conflicts [Terry et al. 1995].

The downside of this technique is that it may detect false conflicts with more than two replicas, as shown in Figure 12. Thus, it is feasible only in systems that employ few sites and experience conflicts infrequently.

### 6.3 Modified-bit algorithm

The modified-bit algorithm, used in the Palm PDA, is a simplification of the two-timestamp algorithm. It works only when the same two sites synchronize repeatedly.

Palm associates each replica (a database record) with a set of bits that tells whether the item is modified, deleted, or archived (i.e., to be deleted from the PDA but kept separately on the PC).

Palm employs two mechanisms, called fast and slow synchronization, to exchange data between a PDA and a PC. Fast synchronization happens in the common case where a PDA is repeatedly synchronized with a particular PC. Here, each side transfers items with the "modified" bit set. A site inspects the attribute bits of each record and decides on the reconciliation outcome — for instance, if it finds the "modified" bit set on both PDA and PC, it marks them as in conflict. This use of "modified" bit can be seen as a variation of two-timestamp algorithm: it replaces $T_i$ with a boolean flag which is set after a replica is modified and reset after the replicas synchronize.

When the PDA is found to have synchronized with a different PC before, the modified-bit algorithm cannot be used. Two sides then revert to the slow mode, in which both ignore the modified bits and exchange the entire database contents. Any item with different values at the two sites are effectively flagged to be in conflict.

### 6.4 Vector clocks and their variations

Vector clocks accurately detect concurrent updates to an object (Section 4.3). Several state-transfer systems use vector clocks to detect conflicts, defining any two concurrent updates

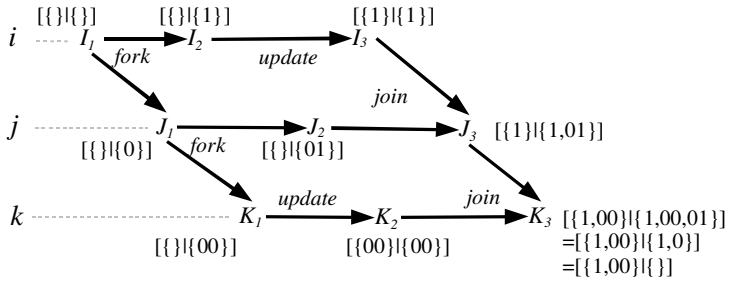



Fig. 13. Example of the use of version timestamps (VTs). An object starts as a single replica $I_1$ with a VT of $[\{\}|\{\}]$. It is forked into two replicas $I_2$ and $J_1$. Site $i$ updates the replica, which becomes $I_3$. Merging replicas $I_3$ and $J_2$ detects no conflict, as $I_3$ dominates $J_2$, as apparent from the fact that $\{1\} \supset \{\}$. In contrast, concurrent updates are detected when merging replicas $J_3$ and $K_2$, as neither of the upd-ids $\{00\}$ and $\{1\}$ subsumes the other.

to the same object to be in conflict. Vector clocks used for this purpose are often called *version vectors* (VV). There is a VV per object per site, and the VVs for different objects are independent from one another.

The LOCUS system introduced VVs and coined the name [Parker et al. 1983; Walker et al. 1983]. Other systems in this category are Coda [Kistler and Satyanarayanan 1992; Kumar and Satyanarayanan 1995], Ficus [Reiher et al. 1994], and Roam [Ratner 1998].

If we consider a single object, its replica at Site $i$ carries a vector clock $VV_i$. $VV_i[i]$ shows the last time an update to the object was submitted at $i$, and $VV_i[j]$ indicates the last update to the object submitted at Site $j$ that Site $i$ has received. The VV is exchanged, updated and compared according to the usual vector clock algorithm (Section 4.3). Conflicts are detected between two sites $i$ and $j$ as follows:

(1) If $VV_i = VV_j$, then the replicas have not been modified.

(2) Otherwise, if $VV_i$ dominates $VV_j$, then $i$ is newer than $j$; that is, Site $i$ has applied all the updates that Site $j$ has, and more. Site $j$ copies the contents and VV from $i$. Symmetrically, if $VV_j$ dominates $VV_i$, the contents and VV are copied from $j$ to $i$.

(3) Otherwise, the operations are concurrent, and the system marks them to be in conflict.

Unlike the two-timestamp algorithm, VVs are accurate: a VV provably detects concurrent updates if and only if real concurrency exists [Fidge 1988; Mattern 1989]. The following two sections describe data structures with similar power to VVs but with different representations.

6.4.1 *Version timestamps.* Version timestamps (VTs) are a technique used in the Panasync file replicator [Almeida et al. 2002; Almeida et al. 2000]. They adapt VVs to environments with frequent replica creation and removal. VT supports only three kinds of operations: *fork* creates a new replica, *update* modifies the replica, and *join(i, j)* merges the contents of replica $i$ into $j$, destroying $i$.

The idea behind VTs is to create a new replica identifier on the fly at fork time, and to merge VTs into a compact format at join time. Figure 13 shows an example of VTs. The VT of a replica is a pair $[upd\text{-}id|hist\text{-}id]$.

*Hist-id* is itself a set of bitstrings that uniquely identifies the history of fork and join operations the replica has seen. An object is first created with a *hist-id* of $\{\}$. After forking, one of the replicas appends 0 to each bitstring in its *hist-id*, and the other appends 1. Thus,

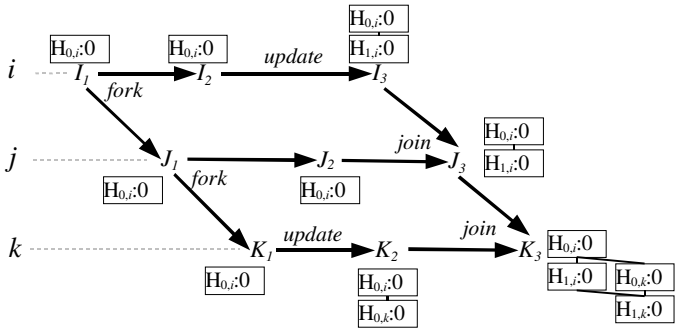



Fig. 14. Example of the use of hash histories (HHes) using the same scenario as Figure 13. The object starts as a single replica on $i$ with a HH of $H_0$, where $H_0$ is a hash of the current contents of the object. After an update at $i$, the HH becomes $H_0$-$H_1$ by appending the new contents hash. The result of merging and resolving two conflicting updates ($K_3$) is represented in the HH by creating an acyclic graph as shown.

the first time an object is replicated after being created, the two replicas have *hist-id*s of $\{0\}$ and $\{1\}$ respectively, whereas replicating $\{00,1\}$ yields $\{000,10\}$ and $\{001,11\}$.

After joining, the new *hist-id* becomes the union of the original two, except that when the set contains two bitstrings of the form $x0$ and $x1$, then they can be merged and contracted to just $x$. Thus, the result of joining replicas $\{0\}$ and $\{1\}$ has *hist-id* $\{0,1\}$ simplified back to $\{\}$; the result of joining $\{001,10\}$ with $\{11\}$ is $\{001,10,11\}$ simplified to $\{001,1\}$.

On the other hand, an *upd-id* simply records the history-id of the replica at the moment when it was last modified.

VTs of replicas of an object precisely capture the happens-before and concurrency relations between them: Site $i$ has seen all updates applied to $j$ if and only if, for each bitstring $x$ in $j$'s *upd-id*, a bitstring $y$ exists in $i$'s *upd-id*, such that $x$ is a prefix of $y$ ($\exists z, y = xz$).

6.4.2 *Hash histories.* Hash histories (HHs) are designed as a dynamic alternative to VCs [Kang et al. 2003]. The basic ideas behind HHs are to (1) record causal dependencies directly by how an object has branched, updated, and merged, and (2) to use a hash of the contents (e.g., MD5), rather than timestamps, to represent the state of a replica. Figure 14 shows an example. While the size of a HH is independent of the number of master replicas, it grows indefinitely with the number of updates. The authors use a simple expiration-based purging to remove old HH entries, similar to the one in Section 6.5.

## 6.5 Culling tombstones

We mentioned in Section 6.1 that the system retains a *tombstone* to mark a deleted object. This is in fact true for any state-transfer system — for instance, when using VVs, the VV is retained as a tombstone. Unless managed carefully, the space overhead of tombstones will grow indefinitely. In most systems, tombstones are erased unilaterally at each site after a fixed period, long enough for most updates to complete propagation, but short enough to keep the space overhead low; e.g., two weeks [Spencer and Lawrence 1998; Kistler and Satyanarayanan 1992; Microsoft 2000]. This technique is clearly unsafe (e.g., a site rebooting after being down for three weeks may send spurious updates), but works well in practice.

Clearinghouse [Demers et al. 1987] lowers the space overhead drastically using a simple technique. In Clearinghouse, tombstones are removed from most sites after the expiration

period, but are retained on a few designated sites indefinitely. When a stale operation arrives after the expiration period, some sites may incorrectly apply that operation. However, the designated sites will distribute an operation that undoes the update and reinstalls tombstones on all other sites.

Some systems rely on a form of commitment algorithm to delete tombstones safely. Roam and Ficus use a two-phase protocol to ensure that every site has received an operation before purging the corresponding tombstone [Guy et al. 1993; Ratner 1998]. The first phase informs a site that all sites have received the operation. The second phase ensures that all sites receive the "delete the tombstone" request. A similar protocol is also used in Porcupine [Saito and Levy 2000]. The downside of these techniques is liveness: all sites must be alive for the algorithm to make progress.

### 6.6 Summary

This section has focused on the specific case of state-transfer optimistic replication systems. Compared to operation-transfer systems, these are amenable to simpler management algorithms, which are summarized in the following table.

| Problem | Solution | Advantages | Disadvantages |
|---|---|---|---|
| Eventual consistency, conflict management. | Thomas's write rule | Simple | Lost updates |
| | Two timestamps | Simple | False-positive conflicts |
| | Modified bits | Simple, space efficient | False-positive conflicts |
| | Vector clock | Accurate conflict detection | Complexity, space |
| Tombstone management. | Expire | Simple | Unsafe |
| | Keep only at designated sites. | Simple | Overhead grows indefinitely at these sites. |
| | Commit | Safe | Complexity, liveness |

## 7. PROPAGATING OPERATIONS

This section examines techniques for propagating operations among sites. A naïve solution exists for this problem: every site records operations in a log, and it occasionally sends its entire log contents to a random other site. Given enough time, this algorithm eventually propagates all operations to all sites, even in the presence of incomplete links and temporary failures. Of course, it is expensive and slow to converge. Algorithms described hereafter improve efficiency by controlling when and which sites communicate, and by reducing the amount of data sent between the sites. Section 7.1 describes a propagation technique using vector clocks for operation-transfer systems. Section 7.2 discusses techniques for state-transfer systems to allow for identifying and propagating only parts an object that have been actually modified. Controlling topology is discussed by Section 7.3. Section 7.4 discusses various techniques for push-based propagation.

### 7.1 Operation propagation using vector clocks

Many operation-transfer systems use vector clocks (Section 4.3) to exchange operations optimally between sites [Golding 1992; Ladin et al. 1992; Adly 1995; Fekete et al. 1997; Petersen et al. 1997]. Figure 15 shows the pseudocode of the algorithm, and Figure 16 shows an example. A Site $i$ maintains vector clock $VC_i$. $VC_i[i]$ contains the number of operations submitted at Site $i$, whereas $VC_i[j]$ shows the number of the last operation,

```
—— Per-site data structures ——
type Operation = record
   issuer: SiteID  // The site that submitted the operation.
   ts: Timestamp  // The timestamp at the moment of issuance.
   op: Operation // Actual operation contents
var vc: array [1 .. M] of Timestamp  // The site's vector clock.
    log: set of Operation // The set of operation the site has received.
—— Called when submitting an operation ——
proc SubmitOperation(update)
   vc[myself] := vc[myself] + 1
   log := log ∪ { new Operation(issuer=myself, ts=vc[myself], op=update)}
—— Sender side: Send operations from this site to site dest ——
proc Send(dest)
   destVC := Receive dest's vector clock.
   upd := { u ∈ log | u.ts > destVC[u.issuer]}
   Send upd to dest.
—— Receiver side: Called via Send() ——
proc Receive(upd)
   for u ∈ upd
      Apply u.
      vc[u.issuer] := max(vc[u.issuer], u.ts)
   log := log ∪ upd
```

Fig. 15. Operation propagation using vector clocks. The receiver-side site first calls the sender's "Send" procedure and passes its vector clock. The sender-side site sends updates to the receiver, which processes them in "Receive" procedure.

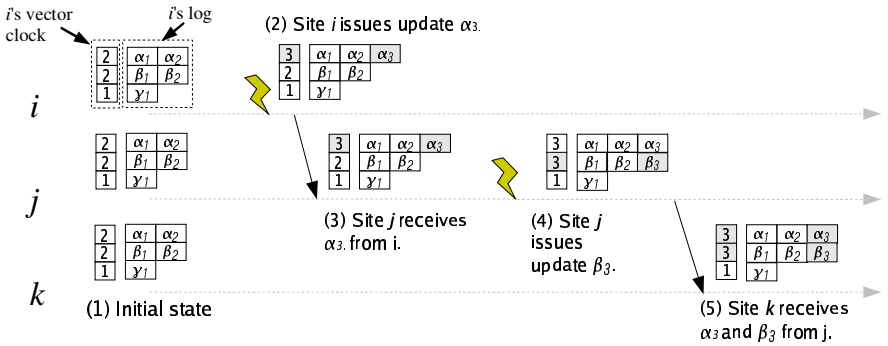



Fig. 16. Example of operation propagation using vector clocks. Symbols α, β and γ show updates submitted at $i$, $j$, and $k$, respectively. Shaded rectangles show changes at each step.

submitted at Site $j$, received by Site $i$. [9] The difference between two VCs shows precisely the set of operations that need to be exchanged to make sites identical.

To propagate operations from Site $i$ to Site $j$, Site $i$ first receives $j$'s vector clock, $VC_j$. Site $i$ compares $VC_i$ and $VC_j$, element by element; if they differ, one site has received (and logged) more operations than the other. For every $k$ such that $VC_i[k] > VC_j[k]$, Site $i$ sends to Site $j$ those operations submitted at Site $k$ that have timestamps larger than $VC_j[k]$. This process ensures that Site $j$ receives all operations stored on Site $i$ and that Site $j$ does not receive the same operation twice.[10] After swapping the roles and letting Site $i$ receive

[9] Alternatively, one could store real-time clock values instead of counters, as done in TSAE [Golding 1992]. $VC_i[j]$ would show the timestamp of the latest operation received by Site $i$ submitted at Site $j$.

[10] Two sites $i$ and $k$ might still send the same operation to $j$ coincidentally. But this duplication should be rare and

operations from Site $j$, the two sites will have received the same set of operations.

## 7.2 Efficient propagation in state-transfer systems

In state-transfer systems, update propagation is usually done by sending the entire replica contents to another, which becomes inefficient as the object size grows. We review several techniques for alleviating this problem without losing the simplicity of state-transfer systems.

7.2.1 *Hybrid state and operation transfer.* Some systems use a hybrid of state and operation transfer. Here, each site keeps a short history of past updates ("diff"s) to the object along with past timestamps recording when these updates were applied. When updating another replica whose timestamp is recorded in the history, it sends only the set of diffs needed to bring it up to date. Otherwise (i.e., if the replica is too old or the timestamp is not found in the history), it sends the entire object contents. Examples include DNS incremental zone transfer [Albitz and Liu 2001], CVS [Cederqvist et al. 2001; Vesperman 2003], and Porcupine [Saito and Levy 2000].

7.2.2 *Hierarchical object division and comparison.* Some systems divide an object into smaller sub-objects. One such technique is to structure an object into a tree of sub-objects (which happens naturally for a replicated file system) and let each intermediate node record the timestamp of the newest update to its children [Cox and Noble 2001; Kim et al. 2002]. It then applies Thomas's write rule on that timestamp and walks down the tree progressively to narrow down changes to the data. Archival Intermemory uses a variation of this idea, called range synchronization, to reconcile a key-value database [Chen et al. 1999]. To reconcile two database replicas, the replicas first compare the collision-resistant hash values of both replicas. If they do not match, then each replica splits the database into multiple parts using a well-known deterministic function, for instance into two sub-databases, one with keys in the lexicographic range of A-L, and the other for the range of M-Z. It then performs hash comparison recursively to narrow down the discrepancies between the two replicas.

Some systems explicitly maintain the list of the names of modified sub-objects and use a data structure similar to vector clocks to detect the set of sub-objects that are modified [Microsoft 2000; Rabinovich et al. 1996]. They resemble operation-transfer systems, but differ in several essential aspects. First, instead of an unbounded log, they maintain a (usually small) list of modified objects. Second, they still use Thomas's write rule to serialize changes to individual sub-objects.

7.2.3 *Use of collision-resistant hash functions.* This line of techniques also divide objects into smaller chunks, but they are designed for objects that lack a natural structure, e.g., large binary files. In the simplest form, the sending side divides the object into chunks, and sends the receiving side a collision-resistant hash value (typically by using SHA-1 or MD5) for each chunk. The receiver requests the contents of every chunk found to be missing on the receiver side. This scheme, however, fails to work efficiently when bytes are inserted or deleted in the middle of the object.

To avoid this problem, the *rsync* file synchronization utility sends hashes in the opposite direction [Tridgell 2000]. The receiving side first sends the hash of each chunk of its

---

is safe.

replica to the sending side. The sender then exhaustively computes the hash value of every possible chunk, at every byte position in the file, and discovers chunks that do not match, and pushes those to the receiver-side replica.

The Low-Bandwidth File System (LBFS) divides objects at boundaries defined by content rather than a fixed chunk size [Muthitacharoen et al. 2001]. The sending side first computes a hash of every consecutive 48-byte sequence, using Rabin's hashing algorithm, which is efficient for this purpose [Rabin 1981]. Every 48-byte sequence that hashes to a particular (well-known but arbitrary) value will constitute a chunk boundary. Then the LBFS sender sends the hash of each chunk to the receiver. The receiver requests only those chunks that it is missing. LBFS reports up to 90% reduction in bandwidth requirements in typical scenarios, over both Unix and Windows file systems. Spring and Wetherall [2000] propose a similar approach for compressing network traffic over slow links.

7.2.4 *Set-reconciliation approach.* Minsky et al. [2001] propose a number-theoretic approach for minimizing the transmission cost for state-transfer systems. This algorithm is applicable when the state of a replica can be represented as a set of fixed-size bitstrings, e.g., hash values. To transmit an object, the sender computes the values of particular polynomial functions from its applied to its set of bitstrings, which it transmits to the receiver. The receiver solves to derive at the exact set of bitstrings it is lacking.

This basic algorithm assumes that the size of the difference between the two sets, $D$, is known *a priori*. It has networking overhead of $O(D)$ and computational complexity of $O(D^3)$. If $D$ is not known *a priori*, the sites can still start from a small guess of $D$, say $D'$. The algorithm can bound the probability of giving false answers given $D$ and $D'$ — thus, one can gradually increase the value of $D'$ until the probability of an error is as low as the user desires. Minsky [2002] proposes a variation of this algorithm, in which the system uses a fixed $D'$. The system recursively partitions the sets using a well-known deterministic function until the $D'$ successfully merges the sub-objects. This algorithm incurs slightly higher networking overhead, but only $O(D')$ computational overhead.

## 7.3 Controlling communication topology

We introduced in Section 3.1 the claim by Gray et al. [1996] that multi-master systems do not scale well, because the conflict rate increases at $O(M^2)$. To derive this result, the authors make two key assumptions: that objects are updated equiprobably by all sites, and that sites exchange updates with uniform-randomly chosen sites. These assumptions, however, do not necessarily hold in practice. First, simultaneous writes to the same data item are known to be rare in many applications, in particular file systems [Ousterhout et al. 1985; Baker et al. 1991; Vogels 1999; Wang et al. 2001]. Second, as we discuss next, choosing the right communication topology and proactively controlling the flow of data will improve propagation speed and reduce conflicts.

The perceived rate of conflicts can be reduced by connecting replicas in specific ways. Whereas a random communication topology takes $O(\log N)$ time to propagate a particular update to all sites [Hedetniemi et al. 1988; Kempe et al. 2001], specific topologies can do better. A star shape propagate in $O(1)$, for instance. A number of actual systems are indeed organized with a central hub acting as a sort of clearinghouse for updates submitted by other masters. CVS is a well-known example (Section 2.5); see also Wang et al. [2001] and Ratner [1998].

*Two-tier replication* is a generalization of the star topology [Gray et al. 1996; Kumar

and Satyanarayanan 1993]. Here, sites are split into mostly connected "core sites" and more weakly connected "mobile sites". The core sites often use a pessimistic replication algorithm to remain consistent with each other, but a mobile site uses optimistic replication and communicates only with the core. Note the difference between single-master systems and multi-master systems with these topologies. The latter types of systems still need to solve the challenges of multi-master optimistic replication systems — e.g., operation scheduling, commitment, and conflict resolution — but they scale better, at the cost of sacrificing the flexibility of communication.

Several other topologies are used in real-world systems. Roam connects core replicas in a ring and hangs other replicas off them [Ratner 1998]. Many choose a tree topology, which combines the properties of both the star and random topologies [Chankhunthod et al. 1996; Yin et al. 1999; Adly 1995; Johnson and Jeong 1996]. Usenet and Active Directory connect sites in a ring or tree structure, supplemented by short-cut paths [Spencer and Lawrence 1998; Microsoft 2000].

In practice, choosing a topology involves a trade-off between propagation speed, load balancing and availability. At one end of the spectrum, the star topology boasts quick propagation, but its hub site quickly becomes overloaded, slowing down propagation in practice; it is also a single point of failure. A random topology, on the other hand, is slower but has extremely high availability and balances load well among sites.

## 7.4 Push-transfer techniques

So far, we have assumed that sites could somehow figure out when they should start propagating to one another. This is not too difficult in services that rely on explicit manual synchronization (e.g., PDA), or ones that rely on occasional polling for a small number of objects (e.g., DNS). In other cases it is better to *push*, i.e., to have a site that with a new operation proactively delivering it to others. This can reduce the propagation delay and eliminates the polling overhead.

7.4.1 *Blind flooding.* Flooding is the simplest pushing scheme. Here, a site with a new operation blindly forwards it to its neighbors. The receiving site uses Thomas's write rule or vector clocks to filter out duplicates. This technique is used in Usenet [Spencer and Lawrence 1998], Active Directory [Microsoft 2000], and Porcupine [Saito and Levy 2000].

Flooding has an obvious drawback: it sends duplicates when a site communicates with many other sites [Demers et al. 1987]. This problem can be alleviated by guessing whether a remote site has an operation. We review such techniques next.

7.4.2 *Link-state monitoring techniques.* Rumor mongering and directional gossiping are techniques for suppressing duplicate operations [Demers et al. 1987; Lin and Marzullo 1999]. Rumor mongering starts like blind flooding, but each site monitors the number of duplicates it has received for each operation. It stops forwarding an operation when the number of duplicates exceeds a limit. In directional gossiping, each site monitors the number of distinct "paths" operations have traversed. An inter-site link not shared by many paths is likely to be more important, because it may be the sole link connecting some site. Thus, the site sends operations more frequently to such links. For links shared by many paths, the site pushes less frequently, with a hope that other sites will push the same operation via different paths.

Both techniques are heuristic and might wrongly throttle propagation for a long timed.

```
—— Global persistent data structures on each site ——
var  log: Set⟨Operation⟩ // The set of operations the site has received.
     tm: array [1 .. N][1 .. M] of Timestamp // The site's timestamp matrix.

—— Sender side: send operations to site dest ——
proc Send(dest)
  ops := ϕ
  for 1 ≤ i ≤ M
    if tm[dest][i] < tm[myself][i] then
      ops := ops ∪ { u ∈ log | u.issuer = i and u.ts > tm[dest][i] }
  Send ops and tm to dest.
  tm[dest] := PairWiseMax(tm[myself], tm[dest])

—— Receiver side: called in response to Send ——
proc Receive(ops, tm_src)
  for u ∈ ops
    if tm[myself][u.issuer] < u.timestamp then
      log := log ∪ { u }
      Apply u to the site
  for 1 ≤ i ≤ N, 1 ≤ j ≤ M
    tm[i][j] := max(t[i][j], tm_src[i][j])
```

Fig. 17. Site reconciliation using timestamp matrices.

For reliable propagation, the system occasionally must resort to plain flooding to flush operations that have been omitted at some sites. Simulation results, however, show that reasonable parameter settings can nearly eliminate duplicate operations while keeping the reliability of operation propagation very close to 100%.

7.4.3 *Multicast-based techniques.* Multicast transport protocols can be used for push transfer. These protocols solve the efficiency problem of flooding by building spanning trees of sites, over which data are distributed. They cannot be applied directly to optimistic replication, however, because they are "best effort" services — they may fail to deliver operations when sites and network links are unreliable. Examples of multicast protocols include IP multicast [Deering 1991], SRM [Floyd et al. 1997], XTP [XTP 2003] and RMTP [Paul et al. 1997].

MUSE is an early attempt to distribute Usenet articles over an IP multicast channel [Lidl et al. 1994]. It solves the lack of reliability of multicast by laying it on top of traditional blind-flooding mechanism — i.e., most of the articles will be sent via multicast, and those that dropped through are send slowly but reliably by flooding. Work by Birman et al. [1999] and Sun [2000] also use multicast in the common case and point-to-point epidemic propagation as a fallback mechanism.

7.4.4 *Timestamp matrices.* A Timestamp Matrix (TM), or matrix clock, can be used to estimate the progress of other sites, in order to push only those operations that are likely to be missing [Wuu and Bernstein 1984; Agrawal et al. 1997]. Figure 17 shows the pseudocode. Site $i$ stores a timestamp matrix $TM_i$, an $N \times M$ matrix of timestamps. $TM_i[i]$ holds $i$'s vector clock (Section 4.3). The other rows of $TM_i$ hold Site $i$'s conservative estimate of the vector clocks of other sites. Thus, if $TM_i[k][j] = t$, then Site $i$ knows that Site $k$ has received operations submitted at Site $j$ with timestamps at least up to $t$. The operation propagation procedure is similar to the one using vector clocks (Section 7.1). The only difference is that the sending Site $i$ uses $TM_i[j]$ as a conservative estimate of Site $j$'s vector clock, rather than obtaining the vector from $j$.

### 7.5 Summary

This section focused on efficient propagation techniques. After briefly considering operation propagation, we mainly discussed improving the efficiency of state propagation in the presence of large objects. Our findings are summarized hereafter.

| System type | Solution | Advantages | Disadvantages |
|---|---|---|---|
| Operation transfer | Whole-log exchange | Simple | Duplicate updates |
| | Vector clocks | Avoids duplicates | $O(M)$ space overhead; complex when sites come and go. |
| State transfer | Hybrid | – | Overhead of maintaining diffs |
| | Object division | – | App-specific, limited applicability. |
| | Hash function | Supports any data type | Computational cost |
| | Set reconciliation | Efficient | Computational cost, limited applicability |
| Push transfer | Blind flooding | – | Duplicate updates |
| | Link-state monitoring | – | Somewhat unreliable |
| | Timestamp matrix | Efficient | $O(M^2)$ space overhead; complex when sites come and go. |

## 8. CONTROLLING REPLICA DIVERGENCE

The algorithms described so far are designed to implement eventual consistency — i.e., consistency up to some unknown moment in the past. They offer little clue to users regarding the quality of replica contents at the present point in time. Many services do fine with such a weak guarantee. For example, replica inconsistency in Usenet is no worse than problems inherent in Usenet, such as duplicate article submission, misnamed newsgroups, or out-of-order article delivery [Spencer and Lawrence 1998].

Many applications, however, would benefit if the service can guarantee something about the quality of replica contents. An example guarantee would be that users will never read data that is more than $X$ hours old. This section reviews several techniques for making such guarantees. These techniques work by estimating some measure of replica divergence and prohibiting accesses to replicas if the estimate exceeds a threshold. Thus, they are not a panacea, as they ensure better data quality by prohibiting accesses to data and decreasing availability [Yu and Vahdat 2001; Yu and Vahdat 2002].

### 8.1 Enforcing read/write ordering

One of the most common complaints with eventual consistency is that a user sometimes sees the value of an object "move backward" in time. Consider a replicated password database [Birrell et al. 1982; Terry et al. 1994]. A user may change her password on one site and later fail to log in from another site using the new password, because the change has not reached the latter site. Such a problem can be solved by restricting when a read operation can take place.

8.1.1 *Explicit dependencies.* The solution suggested by Ladin et al. [1990] and Ladin et al. [1992] is to let the user define the happens-before relationship explicitly for a read operation: an operation specifies the set of update operations that must be applied to the

| Property | Session updated: | Session checked: |
|---|---|---|
| RYW | on write, expand write-set | on read, ensure write-set $\subseteq$ writes applied by site. |
| MR | on read, expand read-set | on read, ensure read-set $\subseteq$ writes applied by site. |
| WFR | on read, expand read-set | on write, ensure read-set $\subseteq$ writes applied by site. |
| MW | on write, expand write-set | on write, ensure write-set $\subseteq$ writes applied by site. |

Table 2. Implementation of session guarantees. For example, to implement RYW, the system updates a user's session when the user submits a write operation. It ensures RYW by delaying a read operation until the user's write-set is a subset of what has been applied by the replica. Similarly, MR is ensured by delaying a read operation until the user's read-set is a subset of those applied by the replica.

replica before the read can proceed. This feature is easily implemented using one of the representations of happens-before introduced in Section 4. For instance, Ladin et al. [1990] represent both a replica's state and an operation's dependency using a vector clock. The system delays the operation until the operation's VC dominates the replica's VC. ESDS follows the same idea, but instead uses a graph representation [Fekete et al. 1999].

8.1.2 *Session guarantees.* A problem with the previous approach is that specifying dependency for each read operation is hard for users. *Session guarantees* are a mechanism to generate dependencies automatically from a user-chosen combination of the following predefined policies [Terry et al. 1994]:

—"Read your writes" (RYW) guarantees that the contents read from a replica incorporate previous writes by the same user.

—"Monotonic reads" (MR) guarantees that successive reads by the same user return increasingly up-to-date contents.

—"Writes follow reads" (WFR) guarantees that a write operation is accepted only after writes observed by previous reads by the same user are incorporated in the same replica.

—"Monotonic writes" (MW) guarantees that a write operation is accepted only after all write operations made by the same user are incorporated in the same replica.

These guarantees are sufficient to solve a number of real-world problems. The stale-password problem can be solved by RYW. MR, for example, allows a replicated email service to retrieve the mailbox index before the email body. A source code management system would enforce MW for the case where one site updates a library module and another updates an application program that depends on the new library module.

Session guarantees are implemented using a *session* object carried by each user (e.g., in a PDA). A session records two pieces of information: the *write-set* of past write operations submitted by the user, and the *read-set* of writes that the user has observed through past reads. Note that each of them can be represented in a compact form using vector clocks. Table 2 describes how the session guarantees can be met using a session object.

## 8.2 Bounding replica divergence

This section reviews techniques that try to bound a quantitative measure of inconsistency among replicas. The simplest are real-time guarantees [Alonso et al. 1990], allowing an object to be cached and remain stale for up to a certain amount of time. This is simple for single-master, pull-based systems, which enforce the guarantee simply by periodic polling. Examples include Web services [Fielding et al. 1999], NFS [Stern et al. 2001], and DNS [Albitz and Liu 2001]. TACT provides real-time guarantees by occasional pushing, see Section 7.4 [Yu and Vahdat 2000].

Other systems provide more explicit means of controlling the degree of replica inconsistency. One such approach is *order bounding*, or limiting the number of uncommitted operations that can be seen by a replica. In the context of traditional database systems, this can be achieved by relaxing the locking mechanism to increase concurrency between transactions. For example, bounded ignorance allows a transaction to proceed, even though the replica has not received the results of a bounded number of transactions that are serialized before it [Krishnakumar and Bernstein 1994]. See also Kumar and Stonebraker [1988], Kumar and Stonebraker [1990], O'Neil [1986], Pu and Leff [1991], Carter et al. [1998] and Pu et al. [1995].

TACT applies a similar idea to optimistic replication [Yu and Vahdat 2001]. TACT is a multi-master operation-transfer system, similar to Bayou, but it adds mechanisms for controlling replica divergence. TACT implements an order guarantee by having a site exchange operations and the commit information (Section 5.5) with other sites. A site stops accepting new updates when the difference between the number of tentative and committed operations exceeds the user-specified limit.

TACT also provides a *numeric bounding* that bounds the difference between the *values* of replicas. The implementation uses a "quota", allocated to each master replica, that bounds the number of operations that the replica can buffer locally before pushing them to a remote replica. Consider a bank account, replicated at ten master replicas, where the balance on any replica is constrained to be within \$50 of the actual balance. Then, each master receives a quota of \$5 ($= 50/10$) for the account. A master site in TACT exchanges operations with other sites. As a side effect, it also estimates the progress of other sites. (TACT uses ack vectors (Section 5.5.2) for this purpose, but timestamp matrices (Section 7.4.4) can also be used.) The site then computes the difference between its current value and the value of another site, estimated from its progress. Whenever the difference reaches the quota of \$5, the site stops accepting new operations and pushes operations to other replicas. Numeric bounding is stronger and more useful than ordering bounding, because it bounds the actual divergence of replica values, although it is more complex and expensive.

## 8.3 Probabilistic techniques

The techniques discussed next rely on the knowledge of the workloads to reduce the replica's staleness probabilistically with small overhead. Cho and Garcia-Molina [2000] study policies based on frequency and order of page re-fetching for web proxy servers, under the simplifying assumption that the update interval follows a Poisson distribution. They find that to minimize average page staleness, replicas should be re-fetched in the same deterministic order every time and at a uniform interval, even when some pages were updated more frequently than others.

Lawrence et al. [2002] do a similar study using real workloads. They present a probabilistic-modeling tool that learns patterns from a log of past updates. The tool selects an appropriate period, say daily or weekday/weekend. Each period is subdivided into time-slots, and the tool creates a histogram representing the likelihood of an update per slot. A mobile news service is chosen as an example. Here, the application running on the mobile device connects when needed to the main database to download recent updates. Assuming that the user is willing to pay for a fixed number of connections per day, the application uses the probabilistic models to select the connection times that optimize the freshness of the replica. Compared to connecting at fixed intervals, their adaptive strategy

shows an average freshness improvement of 14%.

### 8.4 Summary

Beyond eventual consistency, this section has focused on the control of replica divergence over short time periods. The following table summarizes the approaches discussed in this section.

| Problem | Solution | Advantages | Disadvantages |
|---|---|---|---|
| Enforcing causal read & write ordering. | Explicit Session guarantees | – Intuitive | Cumbersome for users. A user must carry a session object. |
| Real-time staleness guarantee. | Polling<br>Pushing | –<br>– | Polling overhead<br>Slightly more complex; network delay must be bounded. |
| Explicit bounding | Order bounding<br>Numerical bounding | –<br>More intuitive. | Not intuitive<br>Complex; often too conservative |
| Best-effort staleness reduction. | Exploit workload pattern. | – | App-specific |

## 9. CONCLUSIONS

This section concludes the paper by summarizing algorithms and systems presented so far and giving hints for designers and users of optimistic replication systems.

### 9.1 Summary of key algorithms and systems

We present a number of tables to summarize the main systems or algorithms mentioned in this survey. Table 3 compares their communication aspects, including the definition of objects and operations, the number of masters, and propagation strategies. Table 4 summarizes the concurrency control aspects of these systems: scheduling, conflict handling, and commitment. Bibliographical sources and cross reference into the text are provided in Table 5.

### 9.2 Comparing optimistic replication strategies

In Table 6, we summarize how different classes of optimistic replication systems compare in terms of the high-level characteristics, including availability, conflict resolution, algorithmic complexity, and space and networking overheads. It is clear that there is no single winner; each strategy has advantages and disadvantages.

Single-master systems are a good choice if the workload is read-dominated or if there is a single writer, because they are simple and free of conflicts.

Multi-master state transfer is reasonably simple, and has a low space overhead (a single timestamp or version vector per object). Its communication overhead is independent of the update rate as multiple updates to the same object are coalesced into a single propagation. The overhead increases with the object size, but it can be reduced substantially, as we discussed in Section 7.2. These systems have difficulty exploiting operation semantics during conflict resolution. Thus, it is a good choice when objects are naturally small, the

| System | Object | Op | *M* | Topology | Propagation | Space reclamation |
|---|---|---|---|---|---|---|
| Active Directory | name-value pair | state | | any | pull | expiration |
| Bayou | | | | | | |
| Bayou (session guarantees) | single DB | op | | any | TV/manual | primary commit |
| Clearinghouse | name-value pair | state | | any | push/pull | expiration |
| Coda | file/directory | both | | star | push | log rollover |
| CVS | file | state | | star | manual | manual |
| Deno | record | op | | any | – | quorum commit |
| DNS | whole DB | state | 1 | tree | push/pull | manual |
| ESDS | arbitrary | op | | any | – | – |
| Ficus, Roam | file/directory | state | | star/ring | pull | commitment |
| IceCube | arbitrary | op | | any | TV/manual | – |
| NIS | whole DB | state | 1 | star | push | manual |
| OT | arbitrary | op | | any | push | – |
| Palm Pilot | DB record | state | | star | manual | – |
| Ramsey & Csirmaz | file/directory | op | | – | – | – |
| TACT | single DB | op | | any | TV/push/pull | primary commit |
| TSAE | single DB | op | | any | TV/push/pull | ack vector |
| Unison | file/directory | op | | any | – | – |
| Usenet | article | state | | any | blind push | expiration |
| Web/file mirror | file | state | 1 | tree | pull | manual |

Table 3. Communication aspects of representative optimistic replication systems. **Op** tells whether the system propagates the object state or semantic operation description. Coda uses state transfer for regular files, but operation transfer for directory operations. *M* stands for the number of masters; it can be any number unless specified. **Topology** shows the communication topology. **Propagation** specifies the propagation protocol used by the system. **Space reclamation** tells the system's approach to delete old data structures. "– " means that this aspect either does not apply, or is not discussed in the available literature.

conflict rate is low, and conflicts can be resolved by a syntactic rule such as "last writer wins".

Multi-master operation transfer overcomes the shortcomings of the state-transfer approach but pays the cost in terms of algorithmic complexity and the space overhead of logging. The networking costs of state and operation transfer depend on various factors, including the object size, update size, update frequency, and synchronization frequency. While state-transfer systems are expensive for large objects, they can amortize the cost when the object is updated multiple times between synchronization.

Table 7 summarizes the key algorithms used to solve the challenges of optimistic replication introduced in Section 3.

### 9.3 Hints for optimistic replication system design

We summarize some of the lessons learned from our own experience and in reviewing the literature.

Optimistic, asynchronous data replication is an appealing technique; it indeed improves networking flexibility and scalability. Some environments or application areas could simply not function without optimistic replication. However, optimistic replication also comes with a cost. The algorithmic complexity of ensuring eventual consistency can be high. Conflicts usually require application-specific resolution, and the lost update problem is ultimately unavoidable. Hence our recommendations:

(1) *Keep it simple.* Traditional, pessimistic replication, with many off-the-shelf solutions,

| System | Ordering | Detecting conflicts | Resolving conflicts | Commit | Consistency |
|---|---|---|---|---|---|
| Active Directory | logical clock | none | TWR | none | eventual |
| Bayou | reception order at primary | predicate | user defined | primary | eventual |
| Bayou (session guarantees) | | | | | ordering |
| Clearinghouse | real-time clock | none | TWR | none | eventual |
| Coda | reception order at primary | vector clock | user defined | primary (implicit) | eventual |
| CVS | primary commit | two timestamps | exclude | primary (implicit) | eventual |
| Deno | quorum | concurrent RW | abort | quorum | 1SR |
| DNS | single master | – | – | – | temporal |
| ESDS | scalar clock | none | none | tacit | 1SR |
| Ficus, Roam | vector clock | vector clock | user defined | none | eventual |
| IceCube | optimization | graph | user defined | primary | eventual |
| NIS | single master | – | – | – | eventual |
| OT | reception order | none | none | tacit | eventual |
| Palm | reception order at primary | modified bits | resolver | primary (implicit) | eventual |
| Ramsey & Csirmaz | canonical | semantic | exclude | – | eventual |
| TACT | reception order at primary | predicate | user-defined | primary | bounded |
| TSAE | scalar clock | none | none | ack vector | eventual |
| Unison | canonical | semantic | abort | primary (implicit) | eventual |
| Usenet | real-time clock | none | TWR | none | eventual |
| Web/file mirror | single master | – | – | – | eventual/ temporal |

Table 4. Concurrency control aspects of some optimistic replication systems. **Ordering** indicates the order the system executes operations. **Detecting conflicts** indicates how the system detects conflicts, if at all, and **Resolving conflicts** how it resolves them. **Commit** is the system's commitment protocol. **Consistency** indicates the system's consistency guarantees. TWRstands for Thomas's write rule.

| System | Main reference | Main Section |
|---|---|---|
| Active Directory | Microsoft 2000 | – |
| Bayou | Petersen et al. 1997 | 2.4 |
| Bayou (session guarantees) | Terry et al. 1994 | 8.1.2 |
| Clearinghouse | Demers et al. 1987 | – |
| Coda | Kistler and Satyanarayanan 1992 | – |
| CVS | Cederqvist et al. 2001 | 2.5 |
| Deno | Keleher 1999 | 5.5.3 |
| DNS | Albitz and Liu 2001 | 2.1 |
| ESDS | Fekete et al. 1999 | 5.5.2 |
| Ficus, Roam | Ratner 1998 | – |
| IceCube | Preguiça et al. 2003 | 5.2.5 |
| NIS | Sun Microsystems 1998 | – |
| OT | Sun et al. 1998 | 5.2.4 |
| Palm Pilot | PalmSource 2002 | 2.3 |
| Ramsey & Csirmaz | Ramsey and Csirmaz 2001 | 5.2.3 |
| TACT | Yu and Vahdat 2001 | 8.2 |
| TSAE | Golding 1992 | 5.5.2 |
| Unison | Balasubramaniam and Pierce 1998 | – |
| Usenet | Spencer and Lawrence 1998 | 2.2 |
| Web/file mirror | Nakagawa 1996 | – |

Table 5. Cross reference

| | **Single master, state transfer** | **Single master, op transfer** | **Multi master, state transfer** | **Multi master, op transfer** |
|---|---|---|---|---|
| **Availability** | low: master single point of failure | | high | |
| **Conflict resolution flexibility** | N/A | | inflexible | flexible: semantic operation scheduling |
| **Algorithmic complexity** | very low | | low | high: scheduling and commitment. |
| **Space overhead** | low: Tombstones | high: log | low: Tombstones | high: log |
| **Network overhead** | O(object-size) | O(\|#operations\|) | O(object-size) | O(\|#operations\|) |

Table 6. Comparing the behaviors and costs of optimistic replication strategies. "Op transfer" stands for operation transfer.

| | **Single Master, state- or op-transfer** | **Multi master, state transfer** | **Multi master, operation transfer** |
|---|---|---|---|
| **Operation propagation** | Thomas's write rule (6.1) | Thomas's write rule, modified bits, version vector(6) | vector clock (4.3) |
| **Scheduling** | Local concurrency control | | Syntactic or semantic (5.2) |
| **Commitment** | | | Operational transformation (5.2.4), ack vector (5.5.2), primary commit (5.5.3), voting (5.5.3) |
| **Conflict detection** | | Two timestamps, modified bits, version vector | Syntactic or semantic |
| **Conflict resolution** | | Ignore, exclude, manual, app. specific (5.4) | |
| **Divergence bounding** | Temporal (8.2), session (8.1.2) | | Temporal, session, numerical, order |
| **Pushing techniques** | Flooding (7.4.1), rumor mongering, directed gossiping (7.4.2) | | Flooding, rumor mongering, directed gossiping, timestamp matrix (7.4.4) |

Table 7. Summary of main algorithms used for classes of optimistic-replication strategies.

is perfectly adequate in small-scale, fully connected, reliable networking environments. Where pessimistic techniques are the cause of poor performance or lack of availability, or do not scale well, try single-master replication: it is simple, conflict-free, and scales well in practice. State transfer using Thomas's write rule works well for many applications. Advanced techniques such as version vectors and operation transfer should be used only when you need flexibility and semantically rich conflict resolution.

(2) *Propagate operations quickly to avoid conflicts.* While connected, propagate often and keep replicas in close synchronization. This will minimize divergence when disconnection does occur.

(3) *Exploit commutativity.* Commutativity should be the default; design your system so that non-commutative operations are the uncommon case. For instance, whenever possible, partition data into small, independent objects. Within an object, use monotonic data structures such as an append-only log, a monotonically increasing counter, or a union-only set. When operations are dependent upon each other, represent the invariants explicitly.

## Acknowledgements

We thank our anonymous reviewers, as well as the following people for their valuable feedback on early versions of this paper: Miguel Castro, Yek Chong, Svend Frølund, Christos Karamanolis, Anne-Marie Kermarrec, Dejan Milojicic, Ant Rowstron, Susan Spence, and John Wilkes.

## REFERENCES

ADLY, N. 1995. *Management of Replicated Data in Large Scale Systems*. Ph. D. thesis, Corpus Cristi College, U. of Cambridge.

ADYA, A. AND LISKOV, B. 1997. Lazy Consistency Using Loosely Synchronized Clocks. In *16th Symp. on Princ. of Distr. Comp. (PODC)* (Santa Barbara, CA, USA, August 1997), pp. 73–82.

AGRAWAL, D., ABBADI, A. E., AND STEIKE, R. C. 1997. Epidemic algorithms in replicated databases. In *16th Symp. on Princ. of Database Sys. (PODS)* (Tucson, AZ, USA, May 1997), pp. 161–172.

ALBITZ, P. AND LIU, C. 2001. *DNS and BIND* (4th ed.). O'Reilly & Associates. ISBN 0-596-00158-4.

ALMEIDA, P. S., BAQUERO, C., AND FONTE, V. 2000. Panasync: Dependency tracking among file copies. In P. GUEDES Ed., *9th ACM SIGOPS European Workshop* (Kolding, Denmark, September 2000), pp. 7–12.

ALMEIDA, P. S., BAQUERO, C., AND FONTE, V. 2002. Version stamps — decentralized version vectors. In *22nd Int. Conf. on Dist. Comp. Sys. (ICDCS)* (Vienna, Austria, July 2002), pp. 544–551. IEEE CS.

ALONSO, R., BARBARA, D., AND GARCIA-MOLINA, H. 1990. Data caching issues in an information retrieval system. *ACM Trans. on Database Sys. (TODS) 15*, 3 (September), 359–384.

BAKER, M., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K., AND OUSTERHOUT, J. K. 1991. Measurements of a distributed file system. In *13th Symp. on Op. Sys. Principles (SOSP)* (Pacific Grove, CA, USA, October 1991), pp. 198–212.

BALASUBRAMANIAM, S. AND PIERCE, B. C. 1998. What is a File Synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)* (October 1998). ACM/IEEE.

BERNSTEIN, P. A. AND GOODMAN, N. 1983. The failure and recovery problem for replicated databases. In *2nd Symp. on Princ. of Distr. Comp. (PODC)* (Montréal, QC, Canada, August 1983), pp. 114–122.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison Wesley. Available online from http://research.microsoft.com/pubs/ccontrol/.

BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. 1999. Bimodal multicast. *ACM Trans. on Comp. Sys. (TOCS) 17*, 2, 41–88.

BIRMAN, K. P. AND JOSEPH, T. A. 1987. Reliable communication in the presence of failures. *ACM Trans. on Comp. Sys. (TOCS) 5*, 1 (February), 272–314.

BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. D. 1982. Grapevine: An exercise in distributed computing. *Comm. of the ACM 25*, 4 (February), 260–274.

BOLINGER, D. AND BRONSON, T. 1995. *Applying RCS and SCCS*. O'Reilly & Associates.

CARTER, J., RANGANATHAN, A., AND SUSARLA, S. 1998. Khazana: An infrastructure for building distributed services. In *18th Int. Conf. on Dist. Comp. Sys. (ICDCS)* (Amsterdam, The Netherlands, May 1998), pp. 562–571.

CEDERQVIST, P., PESCH, R., ET AL. 2001. Version management with CVS. http://www.cvshome.org/docs/manual.

CHANDRA, B., DAHLIN, M., GAO, L., AND NAYATE, A. 2001. End-to-end WAN service availability. In *3rd USENIX Symp. on Internet Tech. and Sys. (USITS)* (San Francisco, CA, USA, March 2001).

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM 43*, 2 (March), 225–267.

CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. 1996. A Hierarchical Internet Object Cache. In *USENIX Winter Tech. Conf.* (San Diego, CA, USA, January 1996), pp. 153–164.

CHARRON-BOST, B. 1991. Concerning the size of logical clocks in distributed systems. *Information Processing Letters 39*, 1 (July), 11–16.

CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. N. 1999. A prototype implementation of archival intermemory. In *Fourth ACM Conf. on Digital Libraries (DL'99)* (Berkeley CA (USA), August 1999), pp. 28–37. ACM.

CHO, J. AND GARCIA-MOLINA, H. 2000. Synchronizing a database to improve freshness. In *Int. Conf. on Management of Data (SIGMOD)* (Dallas, TX, USA, May 2000), pp. 117–128.

CORMACK, G. V. 1995. A Calculus for Concurrent Update. Technical Report CS-95-06, University of Waterloo.

COX, L. P. AND NOBLE, B. D. 2001. Fast reconciliations in fluid replication. In *21st Int. Conf. on Dist. Comp. Sys. (ICDCS)* (Phoenix, AZ, USA, April 2001).

DE TORRES-ROJAS, F. AND AHAMAD, M. 1996. Plausible clocks: constant size logical clocks for distributed systems. In *10th Int. Workshop on Dist. Algorithms (WDAG)* (Bologna, Italy, October 1996).

DEERING, S. E. 1991. *Multicast Routing in a Datagram Internetwork*. Ph. D. thesis, Stanford University.

DEMERS, A. J., GREENE, D. H., HAUSER, C., IRISH, W., AND LARSON, J. 1987. Epidemic algorithms for replicated database maintenance. In *6th Symp. on Princ. of Distr. Comp. (PODC)* (Vancouver, BC, Canada, August 1987), pp. 1–12.

DIETTERICH, D. J. 1994. DEC data distributor: for data replication and data warehousing. In *Int. Conf. on Management of Data (SIGMOD)* (Minneapolis, MN, USA, May 1994), pp. 468. ACM.

ELLIS, C. A. AND GIBBS, S. J. 1989. Concurrency Control in Groupware Systems. In *Int. Conf. on Management of Data (SIGMOD)* (Seattle, WA, USA, May 1989).

ELMAGARMID, A. K. Ed. 1992. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann.

FEKETE, A., GUPTA, D., LUCHANGCO, V., LYNCH, N., AND SHVARTSMAN, A. 1999. Eventually serializable data services. *Theoretical Computer Science 220*, Special issue on Distributed Algorithms, 113–156.

FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. 1997. Specifying and using a partitionable group communication service. In *16th Symp. on Princ. of Distr. Comp. (PODC)* (Santa Barbara, CA, USA, August 1997), pp. 53–62.

FIDGE, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference* (University of Queensland, Australia, 1988), pp. 55–66.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. RFC2616: Hypertext transfer protocol – HTTP/1.1. http://www.faqs.org/rfcs/rfc2616.html.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2, 374–382.

FISCHER, M. J. AND MICHAEL, A. 1982. Sacrificing serializability to attain availability of data in an unreliable network. In *1st Symp. on Princ. of Database Sys. (PODS)* (Los Angeles, CA, USA, March 1982), pp. 70–75.

FLOYD, S., JACOBSEN, V., LIU, C.-G., MCCANNE, S., AND ZHANG, L. 1997. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Journal on Networking 5*, 6 (December), 784–803.

FOX, A. AND BREWER, E. A. 1999. Harvest, Yield, and Scalable Tolerant Systems. In *6th Workshop on Hot Topics in Operating Systems (HOTOS-VI)* (Rio Rico, AZ, USA, March 1999), pp. 174–178.

GIFFORD, D. K. 1979. Weighted voting for replicated data. In *7th Symp. on Op. Sys. Principles (SOSP)* (Pacific Grove, CA, USA, December 1979), pp. 150–162.

GOLDING, R. A. 1992. *Weak-consistency group communication and membership*. Ph. D. thesis, University of California Santa Cruz. Tech. Report no. UCSC-CRL-92-52.

GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. 1996. Dangers of replication and a solution. In *Int. Conf. on Management of Data (SIGMOD)* (Montréal, Canada, June 1996), pp. 173–182.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

GUY, R. G., POPEK, G. J., AND PAGE, T. W., JR. 1993. Consistency algorithms for optimistic replication. In *Proc. First IEEE Int. Conf. on Network Protocols* (October 1993).

HEDETNIEMI, S., HEDETNIEMI, S., AND LIESTMAN, O. 1988. A survey of gossiping and broadcasting in communication networks. *Networks 18*, 319–349.

JAGADISH, H. V., MUMICK, I. S., AND RABINOVICH, M. 1997. Scalable Versioning in Distributed Databases with Commuting Updates. In *13th Int. Conf. on Data Eng. (ICDE)* (Birmingham, U.K., April 1997), pp. 520–531.

JOHNSON, P. R. AND THOMAS, R. H. 1976. RFC677: The maintenance of duplicate databases. http://www.faqs.org/rfcs/rfc677.html.

JOHNSON, T. AND JEONG, K. 1996. Hierarchical matrix timestamps for scalable update propagation. Technical Report TR96-017 (June), University of Florida.

KANG, B. B., WILENSKY, R., AND KUBIATOWICZ, J. 2003. The hash history approach for reconciling mutual inconsistency. In *23rd Int. Conf. on Dist. Comp. Sys. (ICDCS)* (Providence, Rhode Island, USA, May 2003).

KANTOR, B. AND RAPSEY, P. 1986. RFC977: Network news transfer protocol. http://www-.faqs.org/rfcs/rfc977.html.

KAWELL, L., JR., BECKHART, S., HALVORSEN, T., OZZIE, R., AND GREIF, I. 1988. Replicated document management in a group communication system. In *Conf. on Comp.-Supported Coop. Work (CSCW)* (Chapel Hill, NC, USA, October 1988).

KELEHER, P. J. 1999. Decentralized Replicated-Object Protocols. In *18th Symp. on Princ. of Distr. Comp. (PODC)* (Atlanta, GA, USA, May 1999), pp. 143–151.

KEMPE, D., KLEINBERG, J., AND DEMERS, A. 2001. Spatial gossip and resource location protocols. In *33rd Symp. on Theory of Computing (STOC)* (Crete, Greece, July 2001).

KERMARREC, A.-M., ROWSTRON, A., SHAPIRO, M., AND DRUSCHEL, P. 2001. The IceCube approach to the reconciliation of diverging replicas. In *20th Symp. on Princ. of Distr. Comp. (PODC)* (Newport, RI, USA, August 2001).

KIM, M., COX, L. P., AND NOBLE, B. D. 2002. Safety, visibility, and performance in a wide-area file system. In *USENIX Conf. on File and Storage Technologies (FAST)* (Monterey, CA, January 2002). Usenix.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS) 10*, 5 (February), 3–25.

KRASEL, C. 2000. Leafnode: an NNTP server for small sites. http://www.leafnode.org.

KRISHNAKUMAR, N. AND BERNSTEIN, A. 1994. Bounded Ignorance: A Technique for Increasing Concurrency in Replicated Systems. *ACM Trans. on Database Sys. (TODS) 19*, 4 (December), 685–722.

KUMAR, A. AND STONEBRAKER, M. 1988. Semantic based transaction management techniques for replicated data. In *Int. Conf. on Management of Data (SIGMOD)* (Chicago, IL, USA, June 1988), pp. 117–125.

KUMAR, A. AND STONEBRAKER, M. 1990. An analysis of borrowing policies for escrow transactions in a replicated environment. In *6th Int. Conf. on Data Eng. (ICDE)* (Los Angeles, CA, USA, February 1990), pp. 446–454.

KUMAR, P. AND SATYANARAYANAN, M. 1993. Log-based Directory Resolution in the Coda File System. In *2nd Int. Conf. on Parallel and Dist. Info. Sys. (PDIS)* (San Diego, CA, USA, January 1993), pp. 202–213.

KUMAR, P. AND SATYANARAYANAN, M. 1995. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.* (New Orleans, LA, USA, January 1995), pp. 95–106.

LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. 1990. Lazy replication: exploiting the semantics of distributed services. Technical Report TR-484 (July), MIT LCS.

LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. 1992. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys. (TOCS) 10*, 4, 360–391.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM 21*, 7 (July), 558–565.

LAWRENCE, N. D., ROWSTRON, A. I. T., BISHOP, C. M., AND TAYLOR, M. J. 2002. Optimising synchronisation times for mobile devices. In T. G. DIETTERICH, S. BECKER, AND Z. GHAHRAMANI Eds., *Advances in Neural Information Processing Systems*, Volume 14 (2002), pp. 1401–1408. MIT Press.

LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. 2002. Operation shipping for mobile file systems. *IEEE Trans. on Computers 51*, 1410–1422.

LIDL, K., OSBORNE, J., AND MALCOLM, J. 1994. Drinking from the Firehose: Multicast USENET News. In *USENIX Winter Tech. Conf.* (San Francisco, CA, USA, January 1994), pp. 33–45.

LIN, M. J. AND MARZULLO, K. 1999. Directional gossip: Gossip in a wide-area network. In *Third European Dependable Computing Conference* (Prague, Czech, September 1999), pp. 364–379.

LU, Q. AND SATYANARAYANAN, M. 1995. Improving data consistency in mobile computing using isolation-only transactions. In *4th Workshop on Hot Topics in Operating Systems (HOTOS-IV)* (Orcas Island, WA, USA, May 1995).

MATHESON, C. 2003. Personal communication.

MATTERN, F. 1989. Virtual Time and Global States of Distributed Systems. In *Int. W. on Parallel and Dist. Algorithms* (1989), pp. 216–226. Elsevier Science Publishers B.V. (North-Holland).

MAZIÈRES, D. AND SHASHA, D. 2002. Building secure file systems out of Byzantine storage. In *21st Symp. on Princ. of Distr. Comp. (PODC)* (Monterey, CA, USA, July 2002).

MICROSOFT. 2000. *Windows 2000 Server: Distributed Systems Guide*, Chapter 6, pp. 299–340. Microsoft Press, Redmond, WA, USA.

MILLS, D. L. 1994. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM* (London, United Kingdom, September 1994), pp. 317–327.

MINSKY, Y. 2002. *Spread rumors cheaply, quickly and reliably*. Ph. D. thesis, Cornell University.

MINSKY, Y., TRACHTENBERG, A., AND ZIPPEL, R. 2001. Set reconciliation with nearly optimal communication complexity. In *International Symposium on Information Theory* (2001). IEEE.

MISHRA, S., PETERSON, L., , AND SCHLICHTING, R. 1989. Implementing fault-tolerant replicated objects using Psync. In *8th Symp. on Reliable Dist. Sys. (SRDS)* (Seattle, WA, USA, October 1989), pp. 42–53.

MOCKAPETRIS, P. V. 1987. RFC1035: Domain names — implementation and specification. http://www.faqs.org/rfcs/rfc1035.html.

MOCKAPETRIS, P. V. AND DUNLAP, K. 1988. Development of the Domain Name System. In *ACM SIGCOMM* (Stanford, CA, USA, August 1988), pp. 123–133.

MOLLI, P., OSTER, G., SKAF-MOLLI, H., AND IMINE, A. 2003. Safe Generic Data Synchronizer. Rapport de recherche A03-R-062 (May), LORIA.

MOORE, K. 1995. The Lotus Notes storage system. In *Int. Conf. on Management of Data (SIGMOD)* (San Jose, CA, USA, May 1995), pp. 427.

MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. 1995. Exploiting weak connectivity for mobile file access. In *15th Symp. on Op. Sys. Principles (SOSP)* (Copper Mountain, CO, USA, December 1995), pp. 143–155.

MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. 2001. A low-bandwidth network file system. In *18th Symp. on Op. Sys. Principles (SOSP)* (Lake Louise, AB, Canada, October 2001), pp. 174–187.

NAKAGAWA, I. 1996. FTPmirror – mirroring directory hierarchy with FTP. http://noc.intec.co.jp/ftpmirror.html.

O'NEIL, P. E. 1986. The escrow transactional method. *ACM Trans. on Database Sys. (TODS) 11*, 4, 405–430.

Oracle. 1996. *Oracle7 Server Distributed Systems Manual, Vol. 2*. Oracle.

OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, J. A., KUPFER, M. D., AND THOMPSON, J. G. 1985. A trace-driven analysis of the Unix 4.2 BSD file system. In *10th Symp. on Op. Sys. Principles (SOSP)* (Orcas Island, WA, USA, December 1985), pp. 15–24.

PALMER, C. AND CORMACK, G. 1998. Operation transforms for a distributed shared spreadsheet. In *Conf. on Comp.-Supported Coop. Work (CSCW)* (Seattle, WA, USA, November 1998), pp. 69–78.

PALMSOURCE, I. 2002. Introduction to conduit development. http://www.palmos.com/dev/support/docs/.

PARKER, D. S., POPEK, G., RUDISIN, G., STOUGHTON, A., WALKER, B., WALTON, E., CHOW, J., EDWARDS, D., KISER, S., AND KLINE, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Software Engineering SE-9*, 3, 240–247.

PAUL, S., SABNANI, K. K., LIN, J. C., AND BHATTACHARYYA, S. 1997. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications 15*, 3 (April), 407–421.

PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *16th Symp. on Op. Sys. Principles (SOSP)* (St. Malo, France, October 1997), pp. 288–301.

PREGUIÇA, N., SHAPIRO, M., AND MATHESON, C. 2003. Semantics-based reconciliation for collaborative and environments. In *Proc. Tenth Int. Conf. on Cooperative Information Systems (CoopIS)* (Catania, Sicily, Italy, November 2003).

PU, C., HSEUSH, W., KAISER, G. E., WU, K.-L., , AND YU, P. S. 1995. Divergence control for distributed database systems. *Distributed and Parallel Databases 3*, 1 (January), 85–109.

PU, C., HSEUSH, W., KAISER, G. E., WU, K.-L., AND YU, P. S. 1995. Divergence Control for Distributed Database Systems. *Dist. and Parallel Databases 3*, 1 (January), 85–109.

PU, C. AND LEFF, A. 1991. Replica Control in Distributed Systems: An Asynchronous Approach. In *Int. Conf. on Management of Data (SIGMOD)* (Denver, CO, USA, May 1991), pp. 377–386.

RABIN, M. O. 1981. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University.

RABINOVICH, M., GEHANI, N. H., AND KONONOV, A. 1996. Efficient Update Propagation in Epidemic Replicated Databases. In *Int. Conf. on Extending Database Technology (EDBT)*

(Avignon, France, March 1996), pp. 207–222.

RAMAMRITHAM, K. AND CHRYSANTHIS, P. K. 1996. *Executive Briefing: Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society. ISBN 0818674059.

RAMAMRITHAM, K. AND PU, C. 1995. A formal characterization of epsilon serializability. *IEEE Trans. on Knowledge and Data Eng. 7*, 6 (December), 997–1007.

RAMSEY, N. AND CSIRMAZ, E. 2001. An Algebraic Approach to File Synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng. (FSE)* (Austria, September 2001).

RATNER, D., REIHER, P., AND POPEK, G. 1997. Dynamic version vector maintenance. Technical Report CSD-970022 (June), UCLA.

RATNER, D. H. 1998. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. Ph. D. thesis, UC Los Angeles. Tech. Report. no. UCLA-CSD-970044.

RAVIN, E., O'REILLY, T., DOUGHERTY, D., AND TODINO, G. 1996. *Using and Managing UUCP*. O'Reilly & Associates.

REIHER, P., HEIDEMANN, J. S., RATNER, D., SKINNER, G., AND POPEK, G. J. 1994. Resolving File Conflicts in the Ficus File System. In *USENIX Summer Tech. Conf.* (Boston, MA, USA, June 1994), pp. 183–195.

RHODES, N. AND MCKEEHAN, J. 1998. *Palm Programming: The Developer's Guide*. O'Reilly & Associates.

SAITO, Y. AND LEVY, H. M. 2000. Optimistic replication for Internet data services. In *14th Int. Conf. on Dist. Computing (DISC)* (Toledo, Spain, October 2000), pp. 297–314.

SAITO, Y., MOGUL, J., AND VERGHESE, B. 1998. A Usenet performance study. http://www.hpl.hp.com/personal/Yasushi_Saito/pubs/newsbench.ps.

SPENCER, H. AND LAWRENCE, D. 1998. *Managing Usenet*. O'Reilly & Associates. ISBN 1-56592-198-4.

SPREITZER, M. J., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., AND TERRY, D. B. 1997. Dealing with server corruption in weakly consistent, replicated data systems. In *3rd Int. Conf. on Mobile Computing and Networking (MOBICOM)* (Budapest, Hungary, September 1997).

SPRING, N. T. AND WETHERALL, D. 2000. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *ACM SIGCOMM* (Stockholm, Sweden, August 2000).

STERN, H., EISLEY, M., AND LABIAGA, R. 2001. *Managing NFS and NIS* (2nd ed.). O'Reilly & Associates. ISBN 1-56592-510-6.

SUN, C. AND ELLIS, C. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Coop. Work (CSCW)* (Seattle, WA, USA, November 1998), pp. 59–68.

SUN, C., JIA, X., ZHANG, Y., YANG, Y., AND CHEN, D. 1998. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems,. *ACM Transactions on Computer-Human Interaction 5*, 1 (March), 63–108.

SUN, C., YANG, Y., ZHANG, Y., AND CHEN, D. 1996. A consistency model and supporting schemes for real-time cooperative editing systems. In *19th Australian Computer Science Conference* (Melbourne, Australia, Jan 1996), pp. 582–591.

SUN, Q. 2000. Reliable multicast for publish/subscribe systems. Master's thesis, MIT.

SUN MICROSYSTEMS. 1998. Sun Directory Services 3.1 Administration Guide.

TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND

WELCH, B. B. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *3rd Int. Conf. on Parallel and Dist. Info. Sys. (PDIS)* (Austin, TX, USA, September 1994), pp. 140–149.

TERRY, D. B., THEIMER, M., PETERSEN, K., AND SPREITZER, M. 2000. An examination of conflicts in a weakly-consistent, replicated application. Personal Communication.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)* (Copper Mountain, CO, USA, December 1995), pp. 172–183.

THOMAS, R. H. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS) 4*, 2 (June), 180–209.

TRIDGELL, A. 2000. *Efficient algorithms for sorting and synchronization*. Ph. D. thesis, Australian National University.

VESPERMAN, J. 2003. *Essential CVS*. O'Reilly & Associates.

VIDOT, N., CART, M., FERRI'E, J., AND SULEIMAN, M. 2000. Copies convergence in a distributed real-time collaborative environment. In *Conf. on Comp.-Supported Coop. Work (CSCW)* (Philadelphia, PA, USA, December 2000), pp. 171–180.

VOGELS, W. 1999. File system usage in Windows NT 4.0. In *17th Symp. on Op. Sys. Principles (SOSP)* (Kiawah Island, SC, USA, December 1999), pp. 93–109.

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. 1983. The LOCUS distributed operating system. In *9th Symp. on Op. Sys. Principles (SOSP)* (Bretton Woods, NH, USA, October 1983), pp. 49–70.

WANG, A.-I. A., REIHER, P. L., AND BAGRODIA, R. 2001. Understanding the conflict rate metric for peer optimistically replicated filing environments. Submitted for publication.

WESSELS, D. AND CLAFFY, K. 1997. RFC2186: Internet Cache Protocol. http://www.faqs-.org/rfcs/rfc2186.html.

WUU, G. T. J. AND BERNSTEIN, A. J. 1984. Efficient solutions to the replicated log and dictionary problems. In *3rd Symp. on Princ. of Distr. Comp. (PODC)* (Vancouver, BC, Canada, August 1984), pp. 233–242.

XTP. 2003. The xpress transport protocol. http://www.ca.sandia.gov/xtp/.

YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1999. Hierarchical cache consistency in a WAN. In *2nd USENIX Symp. on Internet Tech. and Sys. (USITS)* (Boulder, CO, USA, October 1999), pp. 13–24.

YU, H. AND VAHDAT, A. 2000. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *4th Symp. on Op. Sys. Design and Impl. (OSDI)* (San Diego, CA, USA, October 2000), pp. 305–318.

YU, H. AND VAHDAT, A. 2001. The Costs and Limits of Availability for Replicated Services. In *18th Symp. on Op. Sys. Principles (SOSP)* (Lake Louise, AB, Canada, October 2001), pp. 29–42.

YU, H. AND VAHDAT, A. 2002. Minimal replication cost for availability. In *21st Symp. on Princ. of Distr. Comp. (PODC)* (Monterey, CA, USA, July 2002), pp. 98–107.

ZHANG, Y., PAXON, V., AND SHENKAR, S. 2000. The stationarity of Internet path properties: routing, loss and throughput. Technical report (May), ACIRI.