



Replication: Optimistic Approaches

Yasushi Saito, Marc Shapiro¹
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2002-33
February 8th, 2002*

E-mail: ysaito@hpl.hp.com, Marc.Shapiro@acm.org

optimistic,
replication,
survey

Replication is a key enabling technology in distributed data sharing systems for improving both availability and performance. This paper surveys optimistic replication algorithms, which allow replica contents to diverge in the short term, in order to support concurrent work and to tolerate failures in low-quality communication links. The importance of such techniques is increasing as collaboration through wide-area and mobile networks is becoming more popular.

Optimistic replication algorithms employ techniques vastly different from those for traditional pessimistic algorithms. Whereas a pessimistic algorithm relies on synchronous replica coordination, an optimistic algorithm propagates its updates in the background, discovers conflicts after they happen, and reaches an agreement on the final object contents incrementally. This paper identifies the key challenges that optimistic replication systems face – achieving uniformity, guaranteeing quality of replica contents, and scaling – and presents a comprehensive survey of techniques developed for addressing these challenges.

* Internal Accession Date Only

Approved for External Publication

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

¹ Microsoft Research Ltd., Cambridge (UK)

© Copyright Hewlett-Packard Company 2002

Contents

1	Introduction	4
1.1	Pessimistic Replication Algorithms and their Limitations	4
1.2	Optimistic Replication: Overview and Benefits	6
1.3	Optimistic Replication: Goals and Challenges	7
1.3.1	Uniformity	7
1.3.2	Quality-of-contents guarantee	8
1.3.3	Scalability	9
1.4	Road Map	9
2	Applications of Optimistic Replication	10
2.1	Internet Services	10
2.1.1	DNS: Wide-area Caching and Mirroring	10
2.1.2	Usenet: Wide-area Information Exchange	11
2.2	Mobile Data Systems	11
2.3	CVS: Computer-supported Collaboration	12
2.4	Summary	12
3	Basic Definitions	12
3.1	Objects	12
3.2	Consistency and Conflicts	13
3.3	A Taxonomy of Conflict Handling Techniques	14
4	Single-Master Systems	14
5	Multi-Master State-Transfer Systems	15
5.1	Update Propagation and Conflict Resolution using Thomas's Write Rule	15
5.2	Detecting and Resolving Conflicts in State-transfer Systems	16
5.2.1	Two-Timestamp Algorithm	16
5.2.2	Version Vectors	16
5.2.3	Resolving Conflicts in State-transfer Systems	17
5.3	Supporting Large Objects	17
5.4	Bounding Space Overheads	18
6	Multi-master operation transfer systems	18
6.1	Constructing a log	19
6.2	Update Propagation using Timestamp Vectors	19
7	Scheduling and Conflict Resolution in Operation-Transfer Systems	20
7.1	Syntactic Operation Scheduling	21
7.2	Semantic Operation Scheduling	21
7.2.1	Exploiting commutativity	21
7.2.2	Canonical Operation Ordering	22
7.2.3	Semantic Ordering in IceCube	22
7.3	Operational transformation	23
7.4	Resolving Conflicting Operations	23
7.5	Committing Operations	23
7.5.1	Ack Vectors	24
7.5.2	Primary Commit Protocol	24
7.5.3	Quorum Commit Protocol	24

8	Ensuring Quality of Contents	24
8.1	Preserving Causal Relationship among Accesses	25
8.1.1	Explicit Causal Dependencies	25
8.1.2	Session Guarantees	25
8.2	Bounding Inconsistency	26
8.3	Probabilistic Techniques for Reducing Inconsistency	27
9	Scaling Optimistic Replication Systems	27
9.1	Estimating Conflict Rates	27
9.2	Scaling by Controlling Communication Topology	27
9.3	Push-Transfer Techniques	28
9.3.1	Blind Flooding	29
9.3.2	Link-state Monitoring Techniques	29
9.3.3	Timestamp Matrices: Estimating the State of Remote Sites	29
9.4	Combining Multiple Protocols	30
9.5	Adding and Removing Masters	30
10	Conclusions	31
10.1	Comparing Optimistic Replication Strategies	31
10.2	Summary of Algorithms	31
10.3	Summary of Optimistically Replicated Systems	31
10.4	Hints for Optimistic Replication System Design	31
	Appendix	34
A	The order of events in a distributed system	34
A.1	The Happens-Before Relationship Among Updates	34
A.2	Scalar Clocks	34
A.2.1	Physical Clocks	34
A.2.2	Logical Clocks	34
A.2.3	Counters	34
A.2.4	Limitations of Scalar Clocks	35
A.3	Timestamp Vectors	35

1. INTRODUCTION

Replication maintains *replicas* (copies) of critical data on multiple computers and allows access to any one of them. It is the critical enabling technology of distributed services, improving both their performance and availability. Availability is improved by allowing access to the data, even when some of the replicas or network links are unavailable. Performance is improved in two ways. First, users can access nearby replicas and avoid expensive remote network access and reduce latency. Second, overall service throughput improves by having many sites serve data simultaneously, especially when data accesses are mostly reads. Replication has traditionally been used in local-area networking environments to ensure the availability of mission-critical services, especially database systems and file systems. The evolution of the Internet and mobile computing technologies is changing the picture. Wide-area distributed services are now a practical necessity, as demonstrated by the popularity of services such as DNS, Usenet, CVS, or Lotus Notes. Optimistic replication systems achieve high availability in such environments, where where latency and availability are orders of magnitude worse than in a LAN. They let users read or write any replica at any time, making the service extremely available. In return, they sometimes let replicas carry and present inconsistent data, which is repaired after the fact.

This paper surveys optimistic replication algorithms. We first identify the three key challenges optimistic algorithms face — maintaining data consistency, guaranteeing the quality of contents, and supporting large number of replicas. Next, we analyze previously developed systems and develop a comprehensive taxonomy of algorithms.

The remainder of this section introduces the concept of optimistic replication. Section 1.1 reviews traditional, or “pessimistic” replication algorithms and points out their limitations. Section 1.2 overviews optimistic replication systems at a high level and explains their advantages. Section 1.3 presents the challenges optimistic replication systems face, and Section 1.4 introduces a broad classification scheme we use to review algorithms for solving these challenges and lays the road map for the rest of the paper.

For the reader’s convenience, we summarize recurring notions and terms in Tables 1 and 2. Detailed definitions and examples will be found in the sections referenced.

1.1 Pessimistic Replication Algorithms and their Limitations

Replication has been studied and deployed for a long time. Traditional replication algorithms aim at supporting mission-critical applications and offer *single-copy serializability*. They give users the illusion of having a single, highly available and up-to-date copy of every object. This goal can be achieved in several ways, but the basic concept remains the same: they prohibit access to a replica unless it is provably up to date. We call them “pessimistic” algorithms for this reason. For example, primary-copy algorithms, used widely in commercial database systems [Kronenberg et al. 1986; Oracle 1996; Dietterich 1994; Bernstein and Newcomer 1997], elect one replica as the primary, responsible for serializing requests for a particular object. The other sites act as secondaries that only receive changes from the primary. Other systems, for instance Microsoft Cluster Service [Vogels et al. 1998; Vogels et al. 1998] and DDS [Gribble et al. 2000], serialize updates using two-phase commit, which globally locks all replicas and updates them in unison [Bernstein et al. 1987; Gray and Reuter 1993]. Group multicast systems, e.g., Isis [Birman and Joseph 1987], Totem [Moser et al. 1996], and Transis [Amir 1995], serialize updates by delivering messages to all nodes in the same order. By running a non-replicated database on each node and feeding clients’ requests through such a service, one can in theory build a replicated database system [Patiño-Martinez et al. 2000]. Quorum consensus algorithms make replicas vote every time an object is read or updated, and they require that a plurality of the replicas agree on which content is the newest [Gifford 1979; Thomas 1979; Herlihy 1986; Keleher 1999].

Pessimistic replication techniques perform well in local-area networking environments. Given the continuing progress in Internet technologies, it is tempting to apply these techniques to wide-area

Term	Meaning	Definition
M	Number of master sites	§1.4
N	Number of sites	§1.4
TV	Timestamp vector	§6.2, §A.3
VV	Version Vector	§5.2.2

Table 1. Notations used in the report.

Term	Meaning	Definition
Commit	Agree on a common state or on a final scheduling order.	§6, §7.5
Conflict	Set of updates from different sites that together violate consistency.	§3.2
Consistency	Enforcement of correctness invariants between replicas. (<i>Internal</i> : within an object; <i>external</i> : between objects.)	§3.2
Epidemic replication	Propagation mode that allows any pair of sites to exchange any update.	§1
Log	Record of local, tentative operations.	§6, §6.1
Logical clock	Technique for expressing precedence among updates.	§A.2.2
Master	A replica allowed to issue updates. <i>Single-master</i> : a system supporting one master per object; <i>multi-master</i> : a system supporting several masters per object.	§1.4, §4, §5, §6
Multi-log	Record of all known tentative updates.	§6
Object	Any piece of data being shared.	§3.1
Operation	Semantic description of an update to an object.	§1.4, §6
Operation transfer	Technique that propagates updates as operations.	§1.4, §6
Operational transformation	Technique for transforming the parameters of an operation to make it commute with others.	§7.3
Propagate	Transfer an update to sites that have not seen it yet.	§1
Pull	Technique that makes each site poll others to find new updates.	§5
Push	Technique that makes sites with updates send them to others	§5, §9.3
Physical clock	A hardware clock tracking time on each computer.	§A.2.1
Quality of Contents	The closeness of replicas' contents to each other and to the object's "true" value.	§8
Replica	A copy of an object for local access.	§3.1
Resolve	Detect and repair conflicting updates.	§3.3, §5.2.3, §7
Resolver	An application-provided procedure for resolving conflicts.	§5.2.3
Schedule	The order of applying operations.	§6, §7
State transfer	Technique that propagates entire object contents as an update.	§1.4, §5
Tentative	State or operation applied to a replica while isolated from others. Tentative state or operation might be undone or deactivated before commit.	§1.4, §2.2, §5.1
Timestamp	Any counter that monotonically increases.	§A.2.3
Thomas's write rule	"Last-writer wins" algorithm for state-transfer systems.	§5.1
Timestamp vector	Data structure for tracking the distributed order of operations.	§A.3
Uniformity	Property that replica contents eventually converge	§1.3.1, §5.2.3, §7
Version vector	A per-object timestamp vector, used to detect update conflicts.	§A.3

Table 2. Glossary of recurring terms.

data distribution. Good performance and availability cannot be expected, however, for three main reasons:

- (1) *Wide-area networks continue to be slow and unreliable.* The Internet's end-to-end communication latency and availability have not improved in recent years [Chandra et al. 2001; Zhang et al. 2000]. There are many mobile computing devices and devices behind firewalls with only intermittent connectivity to the Internet.
Pessimistic replication algorithms work poorly in such environments. For example, a primary-copy algorithm demands that a site's failure be accurately detected (and distinguished from, say, link failure or node thrashing) so that it can reliably re-elect another primary when one fails. This is theoretically impossible [Fischer et al. 1985; Chandra and Toueg 1996; Chandra et al. 1996], and becomes probabilistically possible only by over-provisioning hardware end-to-end.
- (2) *Pessimistic algorithms face a trade-off regarding availability:* while increasing the number of replicas improves read availability, it decreases write availability, because these algorithms coordinate replicas in a lock-step manner to update their contents. Thus, they have difficulty supporting services that deploy many replicas and experience frequent updates. Unfortunately, many Internet and mobile services fall in this category; for instance Usenet [Spencer and Lawrence 1998; Lidl et al. 1994], and mobile file and database systems [Walker et al. 1983; Kistler and Satyanarayanan 1992; Moore 1995; Ratner 1998].
- (3) *Some human activities inherently demand optimistic data sharing.* For example, in cooperative engineering or program development, users work concurrently and in relative isolation from one another [Cederqvist et al. 2001; Ferreira et al. 2000]. It is better to allow concurrent updates and repair occasional conflicts after they happen than locking data during editing. Optimistic replication becomes all the more valuable when such working style is combined with network latencies or time-zone differences.

1.2 Optimistic Replication: Overview and Benefits

Optimistic replication algorithms allow users to access any replica at any time, based on the optimistic presumption that conflicting updates are rare, and that the contents are consistent enough with those on other replicas. A key feature that separates optimistic replication algorithms from their pessimistic counterparts is the way updates to data are handled. Whereas pessimistic algorithms update all replicas at once and possibly block "read" requests from users while an update is being applied, optimistic algorithms let any replica be read or written directly most of the time, propagate updates in the background, and reconcile update conflicts after they happen. This feature makes optimistic algorithms more available and efficient using unreliable network media and inexpensive computers. Optimistic replication is not a new idea (e.g., data backup is a crude form of optimistic replication), but its use has recently grown explosively due to the proliferation of the Internet and mobile computing devices. Optimistic replication algorithms offer many advantages over their pessimistic counterparts:

Availability: Optimistic algorithms work well over slow and unreliable network links or computers, because they can propagate updates in the background without blocking accesses to any replicas.

Networking flexibility: Optimistic algorithms also work well over intermittent or incomplete network links. For example, many algorithms allow updates to be relayed through intermediate sites (called *epidemic* communication). Such property is essential in mobile environments in which devices can be synchronized only occasionally.

Scalability: Optimistic algorithms can support a larger number of replicas, because background update propagation demands less coordination among sites.

Site autonomy: Optimistic algorithms often improve the autonomy of sites and users. For example, some services, such as FTP and Usenet mirroring [Nakagawa 1996; Krasel 2000], allow a

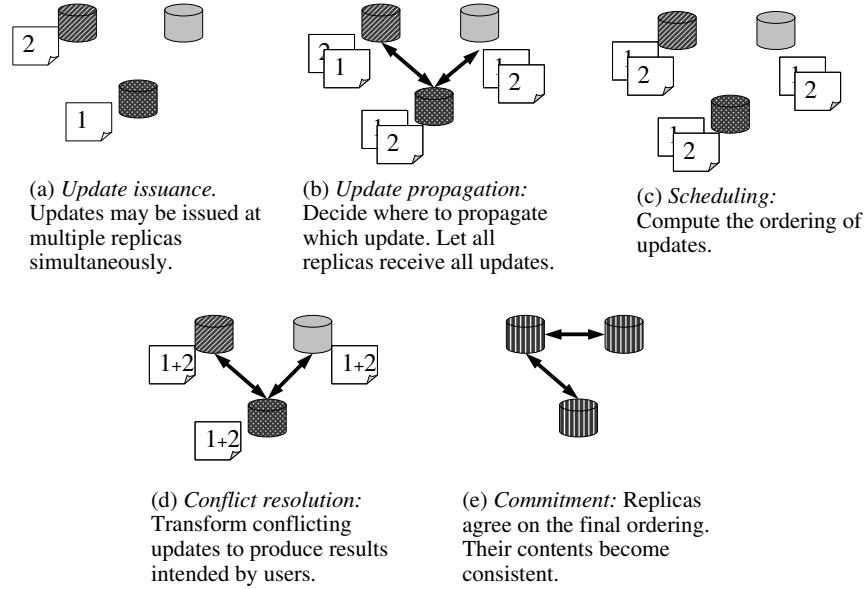


Fig. 1. The steps performed by optimistic replication systems to achieve uniformity.

replica to be added without any administrative change on the existing sites. Optimistic replication is also used in multi-user collaboration, for instance in CVS [Cederqvist et al. 2001; Purdy 2000], PerDiS [Ferreira et al. 2000] and Lotus Notes [Kawell Jr. et al. 1988], to let users work independently on the same project and later merge their changes.

Quick Feedback: Optimistic algorithms usually apply updates as soon as they arrive at a site. This feature lets users see the effects of their updates immediately, resulting in a better user experience. Note, however, that these updates are only *tentative* because the system might undo and redo them when the final application order is determined [Terry et al. 1995; Kermarrec et al. 2001; Gray et al. 1996].

1.3 Optimistic Replication: Goals and Challenges

The aforementioned advantages come with a cost. Any distributed information system faces a trade-off between availability and consistency [Fox and Brewer 1999; Yu and Vahdat 2001]. Pessimistic replication algorithms keep replicas strictly consistent by sometimes prohibiting access to them, whereas optimistic algorithms guarantee any-time access by sometimes letting their contents diverge temporarily: where a pessimistic algorithm waits, an optimistic algorithm speculates.

The high-level goals of an optimistic replication system are to provide access to “sufficiently fresh” data all the time and minimize the user confusion due to divergence. These goals are more precisely stated in terms of the following sub-goals, ordered from the basic to more advanced.

1.3.1 Uniformity. Uniformity, also called *eventual consistency*, demands that replica contents eventually converge, regardless of when and where the updates are issued and in which order they are received at each site. It is the most important goal for optimistic replication systems for two reasons. First, without this guarantee, replica contents may remain corrupted forever. Second, an eventually consistent service usually makes a best effort to disseminate updates quickly among sites, and such a best effort is strong and practical enough for many applications. Uniformity is achieved by the combination of four mechanisms,¹ as depicted in Figure 1: reliable update *propagation* (b),

¹ These mechanisms are often folded into one, as in Thomas’s write rule (Section 5.1).

well-defined update *scheduling* (c), *conflict detection and resolution* (d), and update *commitment* (e).

Update propagation involves a site accumulating changes while being isolated from others, detecting when it can (or should) communicate with another, computing the set of changes to be transferred to the other site to make the two replicas consistent with each other, and transferring the changes quickly.

One desirable property here is *epidemic propagation* [Demers et al. 1987]. Any site communicates with any other and transfers both its own updates and those received from other sites. This protocol eventually broadcasts every update to all sites whatever the network topology, including when connectivity is poor or incomplete, as in Usenet [Spencer and Lawrence 1998]. It is robust when communication links change, as in ad-hoc networking environments (Section 2.2), and when sites fail [Golding 1992; Petersen et al. 1997].

Scheduling defines the update ordering policy. It is needed because different sites may receive the same updates in a different order. A scheduling algorithm has two (somewhat contradictory) goals:

- to apply updates quickly to reduce latency and increase concurrency, and
- to respect user intents and avoid user confusion, e.g., to maintain causal ordering, to avoid conflicts, and not to lose updates.

For example, simply sorting updates totally by order of issuance (e.g., by physical clocks) achieves the uniformity goal, but may increase conflict rate. Taking advantage of application semantics such as commutativity between operation can extract more concurrency, thus improving user experience by minimizing the chance of update conflicts or of the undoing and redoing of tentative updates, at the expense of algorithmic complexity.

Conflict detection and resolution: Conflicts happen, because users can issue updates without knowing of updates issued at other sites. Moreover, the users may not be available when the conflicts are detected. Many systems offer no conflict detection support but still deliver fair results, because people often voluntarily arbitrate accesses to avoid conflicts. For instance, users rarely write to a single file concurrently [Ousterhout et al. 1985; Baker et al. 1991; Vogels 1999; Wang et al. 2001]. Automatic conflict detection, although optional, greatly improves the users' experience. Consider a conference room reservation system for example [Terry et al. 1995]. If two people reserve the same room for the same time slot, the system can easily maintain uniformity by accepting one request and ignoring the other. However, users would be happier if they are informed of the conflict and allowed to negotiate the schedule.

Once a conflict is found, it must be *resolved* by replacing conflicting updates with ones that do not. A common policy, known as “last writer wins”, simply chooses the update with the newest timestamp and discards older updates. A more advanced policy tries to compute a semantic union of the two. For example, in a replicated file system, when two users create files in the same directory, the system can accept both updates if the file names differ. As well, when two users write to the same file simultaneously, the system can create two files with different names, containing the users' versions [Walker et al. 1983; Kistler and Satyanarayanan 1992].

Conflict detection and resolution is usually the most complex part of an optimistic replication system. For this reason, we define conflicts and overview the approaches for remedying them in more detail in Section 3.

Commitment defines a mechanism for making sites agree on the scheduling and the results of conflict resolution. Committed updates can be applied to objects without fear of future undoing. All data structures associated with the updates (e.g., operation log) can be deleted after committed.

1.3.2 Quality-of-contents guarantee. Uniformity, while crucial, guarantees nothing about the actual quality of transient replica contents — that the contents eventually converge means little to users. A quality-of-contents (QoC) guarantee controls when and where updates propagate and replica contents become accessible to users. This feature is not universally adopted, and many systems serve

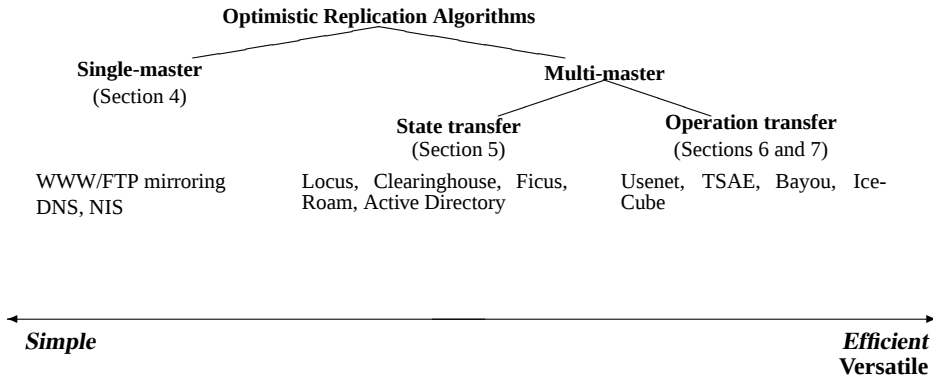


Fig. 2. Taxonomy of optimistic replication algorithms. The summary of the systems mentioned here appears in Table 6.

arbitrarily stale data to users; because it restricts objects accesses, it can lower the system’s availability [Yu and Vahdat 2001].

QoC control takes several forms. The most popular is a *temporal guarantee* that ensures that a replica’s contents are no older than a specified number of seconds from the newest contents, as in WWW caching [Fielding et al. 1999]. *Causal consistency*, preserving causal orderings among read and write requests, is another form of QoC control. For example, 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 avoided by having a *causal read guarantee*, that is, by guaranteeing that a read request honors past updates by the same user. We discuss the QoC issues in Section 8.

1.3.3 Scalability. An optimistic replication system must scale, being able to support large number of replicas and large objects. Having a large number of replicas increases both propagation delay and conflict rate [Gray et al. 1996]. The situation is improved by choosing an appropriate topology linking replicas, and by pro-actively controlling the flow of updates. We discuss these strategies in Section 9. Supporting large object is an issue only for certain type of systems (“state-transfer” systems; see next section); we discuss their strategies in Section 5.3.

1.4 Road Map

The rest of this paper is structured as follows. Section 2 reviews several popular applications that utilize optimistic replication techniques and discusses their diverse requirements. Section 3 defines concepts that play crucial role in this paper: objects, conflicts, and consistency. In particular, we define when updates are said to conflict and overview a range of options to tackle conflicts.

The rest of the paper surveys algorithms for addressing the three challenges mentioned in Section 1.3. Sections 4 to 7 discuss uniformity. We present algorithms for achieving uniformity — e.g., update propagation, scheduling, and conflict resolution — along the taxonomy shown in Figure 2, which classifies the systems by where and how updates are issued and propagated. As we will see, these choices determine types of algorithms that can be used for uniformity maintenance and their functional characteristics, e.g., scalability.

Number of Writers: Single- vs. Multi-Master: The first classification distinguishes where an update can be issued and how it is propagated. *Single-master* systems designate one replica as the *master*. All updates originate at the master and then are propagated to other replicas, or *slaves*. These systems are simple but have limited availability. On the other hand, multi-master

systems let updates be issued at multiple master replicas independently and exchange them in the background. They are more available but significantly more complex. In particular, update scheduling, conflict detection, and resolution are issues unique to multi-master systems. In the paper, we note N the total number of replicas in the system, and M the number of master replicas.

Unit and Mode of Update Transfer: State vs. Operation: The second classification is concerned with *what* is propagated as an update. A *state-transfer* system exchanges the new object contents among sites. An *operation-transfer* system makes each master site remember or reconstruct a *log* of *operations* to the object, and sites exchange operation descriptions. State transfer is simple, because propagating updates only involves transferring the contents of the newest replica to older replicas. Operations transfer is more complex, because sites must agree on the set and schedule of updates. On the other hand, they can be more efficient especially when objects are large and allow for more flexible conflict resolution. For example, in a bibliography database, two updates that modify the authors of two different books can be merged semantically in operation-transfer systems [Golding 1992; Terry et al. 1995], but it is difficult to do the same in a system that transfers the entire database contents every time.

This classification applies to both single- and multi-master systems, but this paper focuses on its implication on the latter type of systems.

Section 4 discusses single-master systems. Section 5 discusses algorithms for multi-master, state-transfer systems. Sections 6 and 7 are devoted to multi-master operation-transfer systems. In particular, Section 6 surveys algorithms from distributing updates, and Section 7 surveys scheduling and conflict handling.

Section 8 discusses the techniques for guaranteeing the quality of replica contents. Three broad approaches are reviewed: preserving causality of read requests, explicitly bounding of inconsistencies, and probabilistic techniques for reducing replica staleness. Section 9 discusses issues scaling optimistic replication as the number of replicas grows. Section 10 concludes this survey by summarizing the algorithms and systems reviewed so far and presenting lessons learned. Appendix A is a brief reminder of techniques for ordering events in a distributed system.

2. APPLICATIONS OF OPTIMISTIC REPLICATION

Optimistic replication is used in several major areas of applications, including wide-area data management, mobile information systems, and computer-based collaboration. To provide some background for the technical discussion that follows, we pick major services in these areas and overview their functions and structures.

2.1 Internet Services

Optimistic replication is attractive when communication between sites is slow and unreliable, which is particularly true for Internet applications. We review two classes of Internet applications, one for broadcasting information, and another for more interactive information exchange.

2.1.1 DNS: Wide-area Caching and Mirroring. Many Internet data dissemination service utilize optimistic replication to ensure availability and performance. Examples include WWW [Fielding et al. 1999; Chankhunthod et al. 1996; Wessels and Claffy 1997; Gwertzman and Seltzer 1996; Wolman et al. 1999], FTP [Nakagawa 1996], NNTP caching [Krasel 2000], 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].

This section overviews DNS, a distributed, hierarchical name service. Names for a particular DNS zone (i.e., sub-tree) 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. To update the database, the

administrator changes the database, increments its timestamp, and reloads it on the master. A slave server periodically polls the master and downloads the database when its timestamp changes.²

DNS is an optimistically replicated service in that it allows the contents on the slaves to lag behind the master's and clients to observe the inconsistency. Optimistic replication is well suited to DNS and similar services for three reasons. First, optimistic replication allows replicating data over unreliable long-haul network links. Second, it avoids the expensive hardware components that would be needed for reliable pessimistic replication. Finally, because DNS already allows stale data to be cached in clients and intermediate servers, using a replication algorithm with loose consistency does not degrade its end-to-end service quality perceptibly. DNS is a single-master (all writes for a zone originate at that zone's master), state-transfer (the master sends a copy of its registration information to slaves) system.

2.1.2 Usenet: Wide-area Information Exchange. Usenet, a wide-area bulletin board system deployed in 1979, is the oldest and still the most 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 non-interactive store-and-forward network [Ravin et al. 1996]. Each server in UUCP could communicate only with a fixed set of neighbors, and network links (usually dial-up phone lines) could support only intermittent file copying between servers directly connected with each other.

Usenet consists of thousands of servers forming a strongly connected (but not complete) directed graph built through a series of human negotiations. Each server replicates all news articles,³ so that a user can read any article from the nearest server. Usenet lets any user post articles to any server. New articles posted on a server are periodically pushed to neighboring servers. A receiving server also stores and forwards these articles periodically to its own neighbors. This way, each article “floods” its way through inter-server links eventually to all servers in the world. Infinite propagation loops are avoided by each server accepting only those articles missing from its disks. An article is deleted from servers either by administratively mandated expiration (e.g., deleting articles more than two weeks old) or by a user posting a cancellation message, which propagates among servers just like ordinary messages but forces servers to delete a particular message.

Usenet's periodic article propagation scheme creates a highly variable delay before an article posted on one server reaches another, sometimes as long as a week. While this variability often confuses users, many accept it to be a reasonable cost to pay for Usenet's excellent availability.

Usenet is an example of a multi-master system (a write can originate at any replica), based on operation transfer (a master sends article creation and cancellation operations to other sites), and using an epidemic update propagation protocol (any site can send an update to any other site).

2.2 Mobile Data Systems

Optimistic replication is especially suited to environments where computers are frequently disconnected. Mobile storage systems use optimistic replication, as in Lotus Notes [Kawell Jr. et al. 1988], Palm [Rhodes and McKeethan 1998], Coda [Kistler and Satyanarayanan 1992; Mummert et al. 1995], Roam [Ratner 1998], and Bayou [Petersen et al. 1997; Terry et al. 1995].

Consider the Bayou system. It allows users to replicate a database on a mobile device (e.g., a laptop computer), modify it while being disconnected, and later merge the changes with any other device that the user manages to communicate with. Bayou remembers changes to a site as a series of high-level operations, and uses timestamp vectors (Section 6.2) to minimize the number of operations exchanged between sites. Updates are applied “tentatively” as they arrive, but the arrival order can

² Recent DNS servers support proactive update notification from the master and incremental zone transfer. See Section 5.3.

³ In practice, articles are grouped into newsgroups, and a server usually stores only a subset of newsgroups to conserve network bandwidth and storage space. Still, articles in a specific newsgroup are replicated on all servers that subscribe to the newsgroup.

be different from the final order on which all sites must agree. Thus, tentative updates are undone and redone repeatedly as the site gradually learns about the final ordering. Conflict detection and resolution are controlled by application-defined scripts that can exploit application semantics.

Algorithmically, Bayou is a multi-master system that uses epidemic propagation of high-level operations. Two types of timestamps control tentative execution and the final schedule, and application-specific scripts resolve conflicts semantically. Bayou is the most sophisticated and complex of the systems mentioned in this section because of the challenging requirements of mobile environments, i.e., slow and unreliable network links, and non-negligible chance of update conflicts.

2.3 CVS: Computer-supported Collaboration

CVS (Concurrent Versions System) keeps a history of changes to a group of files and allows users to retrieve old versions of the files on demand [Cederqvist et al. 2001; Purdy 2000]. The communication in CVS is centralized through a single server. The central server manages the *repository* that contains the authoritative copy of the files along with all changes that happened to them in the past. To update files, a user first creates a private copy (i.e., replica) of the files and edits it using standard tools such as Emacs. Any number of users can have their own private copies and modify them concurrently. After the work is done, the user *commits* her private copy into the repository. If no other user has modified the same file, the commit succeeds immediately. Otherwise, CVS compares the versions line by line. As long as the updates modify disjoint sets of lines, CVS merges them automatically and lets the user commit the merged version. Otherwise, the user is informed of a *conflict*, which she must resolve manually.

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

Although CVS centralizes communication at a single primary server, it is a multi-master operation-transfer system with decentralized conflict detection and resolution.

2.4 Summary

This section overviewed popular systems that use optimistic replication. Optimistic replication has wide-ranging and important applications and has been in use at least since the invention of Usenet. Note the variety of mechanisms these systems use to maintain replica consistency. The following sections of this paper dissect these systems.

3. BASIC DEFINITIONS

This section introduces important vocabulary for the rest of the survey. Section 3.1 defines objects, Section 3.2 defines conflicts, and Section 3.3 classifies techniques for dealing with conflicts.

3.1 Objects

Any replicated system has a concept of a minimal granule of updates, some means of identifying a granule, and a mechanism for propagating an update to a particular granule. We call the minimal granule an *object* in this paper. Objects may be very different between systems. For instance, in a file synchronizer [Balasubramaniam and Pierce 1998], an object is typically either a directory or file. In a groupware system [Palmer and Cormack 1998; Sun 2000], the same file would instead be considered as a document containing many objects; e.g., sections, lines, and characters. Some systems operate at several levels of granularity; e.g., CVS [Cederqvist et al. 2001] operates both at the level of files and lines.

In some systems, object boundaries and identity are context-dependent. For instance, in a collaborative text editor, an object might be a line in a file identified by a line number. As users concurrently modify the file contents, the system needs to compensate for the fact that the same location may be

numbered differently in different contexts. Such relative identity adds considerable complexity to conflict resolution, as demonstrated in the discussion of operational transformations in Section 7.3.

A *replica* is a copy of an object stored at *site*, i.e., a computer. A site may store replicas of multiple objects. This paper, however, sometimes uses *replica* and *site* interchangeably, since most optimistic replication algorithms manage each object independently.

3.2 Consistency and Conflicts

An optimistic replication system lets replicas diverge, i.e., assume different values. In single-master systems, divergence is usually benign, resulting only in a delay of access. Most of this paper is devoted to multi-master systems, where divergence may result in an error, or *conflict*. This section defines more precisely what we mean by conflict in multi-master systems, and its relation to concurrent access and the semantics of shared data.

There are several definitions of conflict suitable for different purposes, but all of them have two things in common. First, a definition of conflict is inseparable from that of consistency, because a conflict occurs when a set of operations violate consistency. Second, an operation that causally “happened-before” another operation (Appendix A.1) never conflicts with each other. In other words, if an operation may have observed the effects of another, systems assume that the former did take the latter into account. From here, the definitions diverge, along two dimensions:

Internal vs. external consistency: Most systems support objects that are independent, discrete, and uniquely identified. Their Independence means that a program that updates several objects cannot assume that the updates are applied atomically (it is not an atomic transaction [Gray and Reuter 1993]). The update to each object is propagated separately, possibly with different conflict resolution outcomes.

Most optimistic replication systems consider objects to be independent, i.e., operations on disjoint objects never conflict. They support *internal* consistency, between replicas of the same object. A typical example is a replicated file system, where updates to different files do not conflict. Other systems try to maintain invariants across objects, or *external* consistency. For instance, a replicated database system uses transactions to ensure that updates to several objects are applied atomically.⁴

Syntactic vs. semantic detection: A system that flags conflicts based only on the occurrence of concurrent updates detects *syntactic* conflicts.⁵ For instance, a file system that flags any two concurrent writes to the same file as conflicting adopts this policy. *Semantic* conflict detection tries to distinguish conflicts from benign concurrent updates using application semantics. For instance, a file system with semantic conflict detection may allow concurrent updates to a directory when they modify different files; creating file “X” does not conflict semantically with deleting file “Y”.

All four possible combinations make sense. Internal-syntactic systems are probably the most prevalent, being very simple to implement. Since internal consistency does not consider relationships between different objects, users may observe surprising results. Consider for instance a persistent hash table where one file contains the table and another the overflow buckets. An insert operation updates both files; concurrent inserts may be resolved differently at the two files, resulting in an incorrect hash table. Nonetheless, support for external consistency is rarely seen in optimistic replication systems, because it necessitates extra mechanisms to maintain or detect inter-object constraints. Some are examined briefly in Section 8. This paper considers internal consistency unless explicitly mentioned otherwise.

⁴ One can emulate multi-object updates by enclosing several small objects into a larger object (Section 5.3). Still, such a system cannot ensure true serializability because of the possibility of concurrent updates. This also restricts object placement, in that a site that replicates any small object must replicate the entire enclosing object.

⁵ Most optimistic replication systems ignore reads.

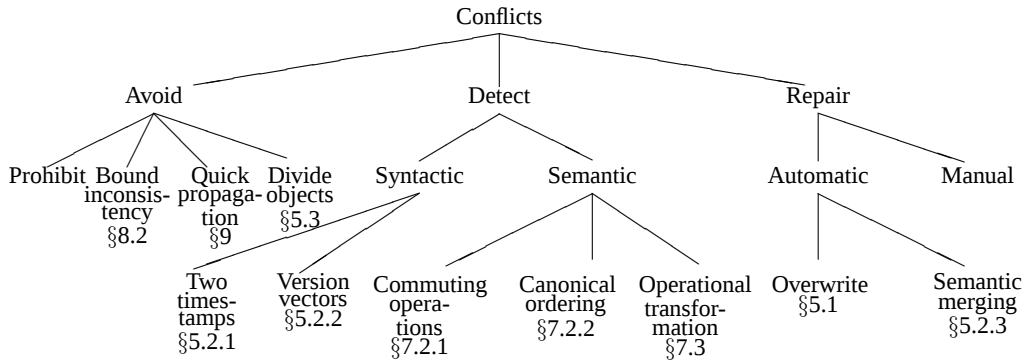


Fig. 3. Taxonomy of conflict handling techniques.

3.3 A Taxonomy of Conflict Handling Techniques

Figure 3 presents a taxonomy of approaches for dealing with conflicts. We distinguish three basic steps: avoid, detect, and repair conflicts.

Conflict avoidance is the best measure, although not always possible in optimistic environments. Pessimistic algorithms (which are out of the scope of this survey) and single-master systems (Section 4) avoid conflicts altogether by serializing updates before propagating. Even when prohibition is impossible, a system can propagate updates quickly to keep replicas from diverging too far (Section 9). Alternatively, dividing objects into small fragments can reduce false-positive conflicts; i.e., updates to different parts of the same object that are not actual conflicts (Section 5.3). Finally, some systems estimate the amount of inconsistency and signal the user when it exceeds a threshold (Section 8.2).

If conflicts are unavoidable, they should be detected and repaired. Syntactic conflicts can be detected using automated mechanisms, such as two-timestamps (Section 5.2.1) and version vectors (Section 5.2.2). Semantic mechanisms can filter out syntactic conflicts that are not conflicting according to object semantics (Section 5.2.3). Examples include detecting commuting updates (Section 7.2.1) and using favorable orderings (Section 7.2.2). Operational transformation modifies parameters of conflicting operations to let them execute in any order even when they are not naturally commutative (Section 7.3).

Conflicting updates must be resolved. Resolution need not depend on detection; one can simply invalidate all updates that can potentially conflict with others, e.g., with a “last writer wins” policy (Section 5.1). Advanced systems support application-specific automatic resolution procedures (Section 5.2.3). In some cases, conflicts are not automatically resolvable, in which case the final alternative is to ask the user or the administrator to resolve the conflict manually.

4. SINGLE-MASTER SYSTEMS

This section discusses the design issues for single-master systems. Single-master systems designate one replica (per object) as the *master* where all updates originate. Update propagation and scheduling in single-master systems is often trivial as data flows only one way, from master to slaves. The simplest design is state transfer (Section 1.4), as in DNS zone transfer [Albitz and Liu 2001], NIS [Sun Microsystems 1998], and WWW/FTP mirroring [Nakagawa 1996]. Here, when the master updates an object, it assigns it a new timestamp. The slave’s timestamp indicates the version it currently holds. A slave polls the master periodically, and if its timestamp differs from the master’s, then it obtains the new contents. Alternatively, the master can push updates to slaves every time the

object is updated, as in DNS proactive pushing [Albitz and Liu 2001], or NIS [Sun Microsystems 1998]. Single-master operation-transfer is also seen in relational database systems [Oracle 1996] and recent implementations of DNS [Albitz and Liu 2001].

Conflict detection and resolution are non-issues in single-master systems, because the master node can immediately detect and delay (or abort) concurrent updates. This property allows single-master systems to potentially scale better when they experience frequent write sharing (Section 9.1).

On the downside, the master becomes the single point of failure and a performance bottleneck. Still, single-master replication is useful when the system's engineering cost must be minimized, or the service is mostly-read.

5. MULTI-MASTER STATE-TRANSFER SYSTEMS

We now commence the discussion of achieving uniformity in multi-master systems. This section focuses on state transfer systems; discussion of operation transfer systems is deferred to Sections 6 and 7.

A state-transfer system involves the following elements at each site: the object contents, and some information describing how up-to-date the replica is (e.g., a timestamp). Replica consistency management is typically carried out in the following steps:

- (1) A site decides to exchange updates with another site. Many systems *pull* updates; i.e., they rely on periodic polling (e.g., FTP mirroring, DNS zone transfer). Other systems, such as NIS, *push* updates; i.e., they make a replica with a new update be responsible for sending it to other replicas. The choice between pulling and pushing is orthogonal to uniformity, but it affects the system's scalability. We discuss the pushing mechanisms in more detail in Section 9.3.
- (2) The two sites find out which objects' contents differ (Section 5.1).
- (3) For each modified object, the sites discover whether it has been modified concurrently. If so, they resolve the conflicts (Section 5.2).
- (4) The resolved state is propagated to the replicas.

We discuss Thomas's write rule, a popular technique that serves both to detect differing replicas and to resolve conflicts. We then introduce several algorithms for more refined conflict detection and resolution. Finally, we discuss scaling issues and space overheads.

5.1 Update Propagation and Conflict Resolution using Thomas's Write Rule

State-transfer systems need to agree only on which replica stores the newest contents, because transferring contents from the newest replica to others will bring all replicas up to date. *Thomas's write rule* is the most popular epidemic algorithm used for this purpose [Birrell et al. 1982; Mockapetris and Dunlap 1988; Rabinovich et al. 1996; Gwertzman and Seltzer 1996]. Its name is derived from a paper that introduced a voting-based pessimistic replication algorithm [Thomas 1979], but it is now used widely in optimistic environments.

Thomas's write rule is shown in Figure 4. Contents managed by Thomas's write rule are always *tentative*; an update can be overridden by a concurrent update by another user, and one cannot tell when an update has been applied to all replicas.

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 *a* updates the object contents (timestamp T_a), and site *b* deletes the object (timestamp T_b) simultaneously. Later, site *c* receives the update from *b* and deletes the replica and timestamp from disk. Site *c* then contacts site *a*. The correct action for *c* is to create a replica when $T_a > T_b$, and ignore the update otherwise. Site *c* cannot decide so, however, because it no longer stores the timestamp.

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 keeps the timestamps (but not the contents)

```

—— Global persistent data structures on each site ——
var data: Contents
    ts: Timestamp
proc IssueUpdate(newData)
    data  $\leftarrow$  newData
    ts  $\leftarrow$  CurrentTime()
    —— Receiver side: called periodically ——
proc ReceiveUpdate(src)
    srcTs  $\leftarrow$  Receive src's timestamp.
    if ts < srcTs then
        data  $\leftarrow$  Receive src's data.
        ts  $\leftarrow$  srcTs

```

Fig. 4. Update propagation using Thomas's write rule. Each object keeps timestamp *ts* that shows the last time it was modified along with *data*, the actual replica contents. An update is issued locally by *IssueUpdate*. Each site calls *ReceiveUpdate* periodically and downloads a peer's contents when its own timestamp is older than the peer's.

of deleted objects on disk, as in Usenet [Spencer and Lawrence 1998], Active Directory [Microsoft 2000], and Clearinghouse [Demers et al. 1987]. Such timestamps are often called “death certificates” or “tombstones”.

5.2 Detecting and Resolving Conflicts in State-transfer Systems

Thomas's write rule implements a “Last writer wins” policy for conflict resolution; it ensures uniformity by making sites eventually converge to the newest value (assuming that communication is complete and updates eventually cease to be generated). This rule, however, does not explicitly detect conflicts, which makes it unattractive in interactive applications that cannot tolerate lost updates. Two techniques have been developed to alleviate this situation: the two-timestamp algorithm and version vectors.

5.2.1 Two-Timestamp Algorithm. The two-timestamp algorithm extends Thomas's write rule to detect syntactic conflicts [Balasubramaniam and Pierce 1998; Gray et al. 1996]. It is used in systems such as Lotus Notes [Kawell Jr. et al. 1988] and Palm Pilot [Rhodes and McKeehan 1998]. Some systems apply the same technique in the context of operation transfer [Terry et al. 1995; Golding 1992].

Here, a replica *i* keeps a timestamp T_i and a *previous* timestamp P_i . T_i is incremented every time an object is updated. Sites *a* and *b* communicate, compare their timestamps, and take the following actions:

- (1) If $T_a = T_b$ and $P_a = P_b$, then the replicas have not been modified.
- (2) If $T_a > T_b$ and $P_a = T_b$, then the object has been modified only on site *a*. Site *b* copies the contents from site *a* and sets T_b, P_b , and P_a to T_a . Symmetrically, if $T_b > T_a$ and $P_b = T_a$, then the contents are copied from *b* to *a*.
- (3) If none of the above holds, there is an update conflict.

The downside of this technique is inaccuracy. It may report false-positive conflicts with more than two masters, as shown in Figure 5. Thus, this technique is feasible only in systems that employ few master sites and experience conflicts infrequently.

5.2.2 Version Vectors. Version vectors (Appendix A.3) is another extension to Thomas's write rule that is more accurate than the previous algorithm. They were first used in the Locus distributed operating system [Parker et al. 1983; Walker et al. 1983].

Here, instead of a single timestamp as in Thomas's write rule, a replica at site *i* carries an *M*-element vector of timestamps VV_i (the VVs for different objects are independent from one another). $VV_i[i]$ shows the last time an update to the object was issued at *i*, and $VV_i[j]$ indicates the last update to the object issued at site *j* that site *i* has received. The VV is exchanged, updated and compared

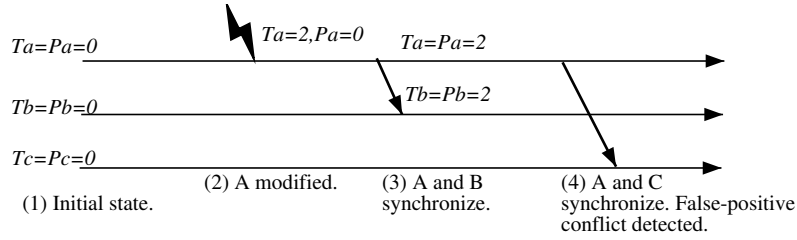


Fig. 5. An example of erroneous conflict detection using the two-timestamp algorithm. T_x and P_x show the current and previous timestamps at site x , respectively. In step (4), sites a and c synchronize, and the algorithm detects a conflict even though site c is strictly older than a .

according to the usual timestamp vector algorithm (Appendix A.3). Conflicts are detected between two sites a and b as follows:

- (1) If $VV_a = VV_b$, then the replicas have not been modified.
- (2) Otherwise, if VV_a dominates VV_b , then the object has been modified only at a . Site b copies the contents and VV from a . Symmetrically, if VV_b dominates VV_a , the contents and VV are copied from b to a .
- (3) Otherwise, there is a (syntactic) update conflict.

Unlike the two-timestamp algorithm, VVs are complete and accurate; a VV provably detects a conflict if and only if a real syntactic conflict exists [Fidge 1988; Mattern 1989]. One downside is that it is not easy to add or remove master replicas, because VVs encode the number of the master sites in itself (see Section 9.5).

5.2.3 Resolving Conflicts in State-transfer Systems. Once a conflict has been detected syntactically using either of the aforementioned algorithms, it must be resolved. Many systems simply store two versions of the object and ask the user to resolve them (e.g., Lotus [Kawell Jr. et al. 1988] and Palm Pilot [Rhodes and McKeehan 1998]).

Sometimes, it is possible to automate resolution by exploiting the object semantics. Replicated file systems such as Locus [Walker et al. 1983], Ficus, Roam [Reiher et al. 1994; Ratner 1998], and Coda [Kumar and Satyanarayanan 1995] incorporate semantic detection and resolution via *resolver* programs for well-known file types. For instance, a syntactic conflict on a mail folder file can be resolved by computing the union of the messages from the conflicting replicas. On the other hand, a conflict on compiled files can be left unresolved, because they can be regenerated from their source.

Writing a useful automatic resolver is not a trivial task. First, semantic resolution in state-transfer systems is limited, as the resolver can only access the states of the conflicting replicas. For instance, the above mail resolver may cause a message deleted in one of the replicas to re-appear. Second, the resolver must obey strict rules to ensure uniformity. It must be deterministic; rules for choosing and running a resolver must be identical at every resolving site. A resolver may depend only on the current state [Petersen et al. 1997] and not access time-varying or site-specific information such as the current system clock or the site's name. Resolvers must commute; i.e., given a set of conflicting updates, resolution must produce the same result regardless of the order they arrive at a site. Finally, resolvers that fail due to exceeding their limits on resource usage must fail deterministically: all sites must place uniform bounds on the CPU, memory, and file resources allocated to a resolver, and must consistently enforce these bounds during execution.

5.3 Supporting Large Objects

The primary scaling bottleneck in state-transfer systems is that the size of an update (and the propagation overhead) increases with object size. This problem can be alleviated in several ways without

losing the simplicity of state-transfer systems.

Some systems use a hybrid of state- and operation-transfer; for instance, DNS incremental zone transfer [Albitz and Liu 2001], CVS [Cederqvist et al. 2001; Purdy 2000], and Porcupine [Saito and Levy 2000]. Here, each site keeps a short history of past updates (“diff”s) to the object along with past timestamps recorded when these updates were applied. When updating another replica whose timestamp is recorded in the history, it sends only the 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. This technique is far simpler than operation transfer, because it uses a single timestamp to serialize updates.

Another approach is to divide an object into smaller sub-objects.⁶ The key issue here is efficiently discovering newly updated sub-objects, because simply polling each sub-object independently is too costly. Some systems structure sub-objects into a tree (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]. They then apply Thomas’s write rule on that timestamp and walk down the tree progressively to narrow down changes to the data. Other systems explicitly maintain the log of updates to sub-objects and use a data structure similar to version vectors 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, the log size is naturally small and bounded, because they only log the names of modified sub-objects, and there is only a finite number of sub-objects. Second, they still use Thomas’s write rule to serialize individual sub-objects.

5.4 Bounding Space Overheads

Tombstones are the main source of space overhead in state-transfer systems (VV-based systems still need to keep VVs as tombstones). While tombstones are small, they can fill the disk up when objects are created and deleted frequently. In most systems, tombstones are erased unilaterally after a fixed period, long enough for most of updates to complete propagation, but short enough to keep the space overhead of tombstones low; e.g., one month [Spencer and Lawrence 1998; Kistler and Satyanarayanan 1992; Microsoft 2000]. This technique is clearly unsafe (e.g., a site rebooting after being down for two months may send spurious updates), but works well in practice.

Some systems run consensus algorithms to delete tombstones safely. Roam and Ficus use a two-phase protocol to ensure that every site has received an update before purging the corresponding tombstone [Guy et al. 1993; Ratner 1998]: Phase one informs a site that all sites have received the update, and phase two ensures that all sites receive the “delete the tombstone” request. A similar protocol is also used in Porcupine [Saito and Levy 2000].

6. MULTI-MASTER OPERATION TRANSFER SYSTEMS

We now turn to the study of multi-master operation-transfer systems. Operation-transfer systems are substantially more complex than state-transfer systems, because replicas need to agree on the set and order of operations just to achieve the uniformity goal. An operation-transfer system works basically as follows:

- (1) Replicas start in some common initial state.
- (2) Each master replica maintains a *log* of locally submitted update operations. In addition, every replica (be it a master or not) collects operations logs from all master replicas into what we call a *multi-log*. Section 6.1 discusses logging algorithms in detail.
- (3) A master accepts update operations from the local user, applies them locally, and adds them to the log and the multi-log. A replica also receives operations from remote masters, applies them, and adds them to the multi-log. Section 6.2 discusses update propagation in more detail.

⁶ This has the added advantage of providing external consistency between the sub-objects.

- (4) The correct update application order, called the *schedule*, is learned by replicas only gradually and is often different from the update reception order. When a site adds a request to the multi-log, it recomputes the schedule (to the best of its knowledge) and plays the schedule, possibly undoing and reapplying operations already applied to the replica, to reach the next tentative state. The beginning of Section 7 (up to Section 7.3 inclusive) discusses scheduling algorithms.
- (5) Optionally, when operations are found to be conflicting, replicas resolve them. The beginning of Section 7 discusses conflict handling.
- (6) When all sites agree on the schedule for a set of operations, they *commit* the state, remove the records for these operations from the multi-log, and repeat the algorithm from the start. Section 7.5 discusses commit algorithms.

6.1 Constructing a log

A log can be viewed as a program for bringing the shared objects from an initial state to a tentative state. There are three choices when designing a logging algorithm:

- The layer of abstraction where logging takes place.
- Whether to clean redundant actions.
- Whether to record or to reconstruct user actions.

The first choice is the level of abstraction of the logged operations. At one end of a spectrum, a log might contain only reads and writes to byte regions, which provide little semantic content. At the other end, a log might record high-level user actions; this is both concise and expressive but complex to reason about. Ramsey and Csirmaz [2001] suggest to log simple operations (simple to reason about) grouped to represent high-level intents (expressive).

A log that contains no redundant actions is said to be *clean*; no shorter log achieves the same tentative state from the same initial state. A clean log is not just more efficient [Blaustein and Kaufman 1985; Berger et al. 1996; Dedieu and Pacitti 2001], but also easier to reason about [Ramsey and Csirmaz 2001]. Consider a collaborative jigsaw puzzle as an example Kermarrec, Rowstron, Shapiro, and Druschel [2001]. Suppose user A puts piece X in some position. Concurrently, user B puts the same piece X in a different position, then changes his mind and removes it. If B's log is clean, it contains no moves concerning X, and the system can easily determine that the operations are conflict-free.

The final choice is about whether to record operations as they are executed, or to reconstruct a possible history by “diffing” (comparing) the initial and the current state. Recording can log operations at any abstraction level by intercepting appropriate user activities, but it must implement a separate cleaning pass. In contrast, diffing can be implemented outside of the object editor, and its log is naturally clean; but it involves intimate knowledge of the editor's semantics.

6.2 Update Propagation using Timestamp Vectors

Operation-transfer systems must propagate all updates to all sites, even if not all sites are fully connected at any one time (i.e., epidemic update propagation, defined in Section 1). A naïve solution exists for this problem: let each site periodically send its entire multi-log to the other sites. Given enough time for sites to communicate at least indirectly, this algorithm eventually propagates all operations to all sites. It has an obvious downside, that is, unbounded disk space and network bandwidth consumption as it accumulates more updates.

In an operation-transfer system, the solution is to use timestamp vectors (Appendix A.3) [Golding 1992; Ladin et al. 1992; Adly 1995; Fekete et al. 1996; Petersen et al. 1997]. Site i records an M-element timestamp vector TV_i . $TV_i[i]$ keeps the number of operations issued at i , i.e., the size of its log.⁷ $TV_i[j]$ represents the newest update issued at site j received by site i . The difference between

⁷ In TSAE (Time-Stamped Anti-Entropy protocol) [Golding 1992], $TV_i[i]$ keeps the last (physical-clock) time an update was

```

—— Global persistent data structures on each site ——
const  $M$  = Number of master sites.
type Update = record
  issuer: SiteID // The site that issued the update.
  ts: Timestamp // The timestamp at the moment of issuance.
  op: Operation // Actual update contents
end

var tv: array [0 ..  $M-1$ ] of Timestamp // The site's timestamp vector.
    multiLog: Set(Update) // The set of update the site has received.

proc IssueUpdate(val)
  tv[myself]  $\leftarrow$  tv[myself] + 1
   $u \leftarrow$  new Update(issuer=myself, ts=tv[myself], op=val)
  multiLog  $\leftarrow$  multiLog  $\cup$  {  $u$  }

—— Sender side: Send updates from this site to site dest ——
proc SendUpdates(dest)
  destTV  $\leftarrow$  Receive dest's timestamp vector.
  upd  $\leftarrow$   $\emptyset$ 
  for  $0 \leq i < M$ 
    if destTV[i] < tv[i] then
      upd  $\leftarrow$  upd  $\cup$  {  $u \in$  multiLog |
         $u.issuer = i \wedge u.ts > destTV[i]$  }
  Send upd to dest.

—— Receiver side: Called via SendUpdates() ——
proc ReceiveUpdates(upd)
  for  $u \in$  upd
    Apply  $u$  to the site
    tv[u.issuer]  $\leftarrow$  max(tv[u.issuer], u.ts)
  multiLog  $\leftarrow$  multiLog  $\cup$  upd

```

Fig. 6. Replica reconciliation using timestamp vectors. The receiver-side site first calls the sender's "SendUpdate" procedure and passes its timestamp vector. The sender-side site sends updates to the receiver, which processes them in "ReceiveUpdates" procedure.

two TVs shows precisely the set of updates that need to be exchanged to make them identical.

Figure 6 shows the update propagation procedure using timestamp vectors. To propagate updates from site i to site j , site i first receives j 's timestamp vector, TV_j . Site i compares TV_i and TV_j , element by element. If $\exists k, TV_i[k] > TV_j[k]$, site i extracts from its multi-log those updates issued at site k , stored on i , and with timestamps larger than $TV_j[k]$, and sends them to site j . This process ensures that site i receives all updates stored on site j and that site i never receives any update that it has already received. After swapping the roles and letting site j receive updates from site i , the two sites will have received the same set of updates.

7. SCHEDULING AND CONFLICT RESOLUTION IN OPERATION-TRANSFER SYSTEMS

This section reviews algorithms for scheduling operations in an operation-transfer system and resolving conflicting operations to achieve uniformity. Scheduling is closely related to reliable group communication that delivers network messages to sites in a well-defined order [Birman and Joseph 1987; Birman et al. 1991; Hadzilacos and Toueg 1994]. In fact, TSAE (Time-Stamped Anti-Entropy protocol) [Golding 1992] is presented as a group communication service, although it is more easily understood as an optimistic replication service. Syntactic update ordering presented in this paper roughly corresponds to total and causal broadcast, and semantic ordering corresponds to causal broadcast with an application-specific policy to resolve concurrent updates. However, the mechanisms for implementing such policies are very different, because optimistic replication systems work in an asynchronous, epidemic way.

Starting from the same initial state and given the same set of operations, uniformity in operation-

locally issued. The update propagation process works identically otherwise.

transfer systems is achieved when each replica computes a schedule that produces an identical final state. More precisely, we require the following properties:

- (1) There exists an equivalent prefix of the schedules at all sites, which we call the *committed* prefix. (Schedule prefixes are equivalent when they contain the same operations, and they produce the same state starting from a same initial state.)
- (2) Every operation is eventually included in the committed prefix, with a status of either *activated* (executed) or *deactivated* (not executed).
- (3) Once included in the committed prefix, an operation is never removed, and never changes status.

To allow for conflict resolution, the above definition allows a schedule to deactivate one or more operations, i.e., not execute them. Thus, if two operations conflict, one or the other is deactivated. A more general definition would allow a set of conflicting operations to be transformed into a non-conflicting set.

There are several possible approaches for generating an equivalent schedule. A sufficient condition is that the schedules are identical (assuming deterministic operations). *Syntactic* scheduling strategies (Section 7.1) use static rules to order updates totally and generate an identical schedule. An identical schedule, however, is not a necessary condition. *Semantic* scheduling (Section 7.2) exploits operation semantics to order updates partially to reduce conflicts and yet generate equivalent schedules. Operational transformation, described in Section 7.3, modifies operations themselves to make schedules equivalent without reordering them. Section 7.4 discusses strategies for resolving conflicting updates. Section 7.5 discusses update commitment (or stabilization), which is a mechanism for letting replicas agree on a schedule and allowing updates to be permanently applied to the replicas without fear of future undoing.

7.1 Syntactic Operation Scheduling

The minimal requirement for operation scheduling is that the schedule be compatible with the happens-before partial order (Appendix A.1). Syntactic scheduling policies provide this minimal requirement via static ordering rules.

Many systems record a scalar clock (i.e., physical clock or logical clock; see Appendix A.2) for each operation and use that to order operations, as in Bayou [Petersen et al. 1997] and TSAE [Golding 1992]. Since a scalar clock alone cannot detect conflicts, these systems must use a separate mechanism to detect conflicts semantically.

One could imagine using version vectors (Section 5.2.2) for update ordering and conflict detection, but they are not common in operation-transfer systems. The reason is that the multi-log usually already capture the semantic relationships between the operations, and version vectors offer little additional value.

7.2 Semantic Operation Scheduling

A different approach to operation scheduling is to view scheduling as an optimization problem for reducing conflicts and achieving uniformity. Such *semantic scheduling* is more complex than syntactic scheduling but may reduce conflicts and reordering of tentative operations.

7.2.1 Exploiting commutativity. The simplest semantic scheduling exploits commutativity between operations. Two operations known to commute a priori can be scheduled in either order without impacting the final state [Pu and Leff 1991]. This property has long been exploited in database systems, in which query optimizers often reorder commutative operations to generate a more efficient schedule [Date 2000, Chapter 17]. Wu and Bernstein [1984] exploit commutativity to maintain a multi-master, replicated log data structure.⁸ In such a data structure, both insertion and deletion are

⁸ This log is an application-level data structure, unrelated to the operation log used for update propagation.

commutative and idempotent (assuming that each new record is unique), and scheduling becomes a non-issue. In GemStone [Almarode and Bretl 1998], an object type can inform the database engine about commutativity between its methods (operations). The database detects possible syntactic conflicts and ignores those that belong to operations declared commutative by the data type.

7.2.2 Canonical Operation Ordering. Canonical ordering is a technique for defining operation ordering formally from application semantics. It was pioneered by Ramsey and Csirmaz [2001], who formalized the semantics of optimistic replication and reconciliation in a file system. Here, logs and scheduling are ordered using a formal algebra that precisely defines how operations interact. For instance, if an operation deletes a directory, and another operation (in another log) deletes one of its subdirectories, then the subdirectory must be deleted before the parent. When there is no natural structural order, the canonical order is arbitrary. Two operations from different logs (semantically) conflict either if they do not commute, or if the combination would “break” the file system structure. This might happen, for instance, if one user modifies a file, and another deletes its parent directory.

Scheduling orders multi-logs so that replicas of the shared file system become “as close as possible.” The scheduler ignores duplicate updates. It deactivates (does not apply) conflicting operations and those depending on previously deactivated operations, saving them for manual resolution.

The operations supported by Ramsey and Csirmaz [2001] are very few and primitive: create, remove, edit.⁹ Nonetheless, they list an impressive 51 rules for their algebra. The applicability of this technique in more complex environments is an open issue.

7.2.3 Semantic Ordering in IceCube. IceCube is an application-independent replication toolkit that computes the optimal scheduling of a multi-log from semantic constraints on operations [Ker-marrec et al. 2001; Preguiça and Shapiro 2001]. IceCube supports two types of constraints:

- (1) A *static constraint* relates two operations without a reference to the state of the modified objects. Among them, a *log constraint* specifies the relation between operations in a single log; i.e., it expresses an intent of the user. For example, a log constraint may impose an execution order of two operations (*predecessor-successor* constraint), or it may specify an atomic unit of update application (*parcel* constraint), or it may specify a set of alternative operations from which the scheduler can choose (*alternatives* constraint).
An *object constraint* relates operations that modify shared objects. Four types of object constraints are defined: overlapping (the two operations update the same object), commutative, mutually exclusive, and best-order (one execution order is more favorable than the other).
- (2) A *dynamic constraint* is an assertion on the schedule that must evaluate to true at execution time. This can be either Bayou-style “dependency checks” (Section 7.4) or classical pre- and post-conditions.

As an example, consider a travel-planning application that can run disconnected using IceCube. The user might reserve a flight, pay for his ticket, and plan to hold a meeting on arrival. During the same session, the user also sets an unrelated meeting. When the user reconnects, a conflict may appear due to the flight being full, insufficient funds, or a conflicting meeting.

The user indicates his intents by grouping the three related operations in a parcel, indicating that if any of them fails they should all be reconciled consistently. The unrelated meeting is not made part of the same parcel. If different flights are equally desirable, they are indicated as alternatives. The shared calendar implements its semantics by making incompatible meeting requests mutually exclusive. According to bank account semantics, credits commute with each other, debits with one another, and credits are best-ordered before debits. Finally, the bank sets a dynamic constraint that the account used to pay for the ticket is not allowed to go negative.

⁹In particular, there is no “move”, because such an operation would modify two directories, the source and the destination; instead a move is modeled as a create and a remove.

The IceCube scheduler repeatedly selects and tries out a schedule that fits the static constraints. If a dynamic constraint is not satisfied, IceCube aborts execution of the current schedule and proposes another one. Using static heuristics that give preferential treatment to operations most likely to succeed, according to the static constraints and to the recent history of dynamic constraint failures, IceCube can schedule a few thousand operations in seconds [Preguiça and Shapiro 2001].

7.3 Operational transformation

Operational transformation (OT) aims to ensure uniformity even when schedules differ, operations are non-commutative, and they cannot be reordered or rolled back [Sun and Ellis 1998]. The canonical application of OT is a cooperative text editor where an operation specifies the character position of a change [Cormack 1995; Vidot et al. 2000]. Suppose that users edit a shared string “abc”. User 1 executes `insert(“X”,1)` yielding “Xabc”, and sends the update to Site 2. User 2 executes `delete(1)` yielding “bc”, and sends the update to Site 1. The result at Site 2 is “Xbc” as expected, but the result at Site 1 is an unexpected “abc”. OT modifies the parameters of the operation to preserve its intention and make its effect identical on all sites, regardless of reception order. In this example, OT detects that the insert and the delete are concurrent, and it modifies the delete operation at Site 1 to be `delete(2)`.

OT is complex and highly dependent on application semantics. While transforming a pair of concurrent operations is relatively intuitive, things get much more complicated with three or more concurrent threads [Sun and Ellis 1998]. It is possible to reason formally about this issue using the calculus of Cormack [1995]. For example, Palmer and Cormack [1998] prove the correctness of transformations for a shared spreadsheet example, supporting all the usual operations such as updating cell values, adding or deleting rows or columns, and changing formulae.

7.4 Resolving Conflicting Operations

Conflict resolution is, by nature, highly application-specific and difficult to generalize. This is especially true for operation-transfer systems that usually describe operations semantically. This section takes Bayou as an example and overviews its attempt for separating application logic from reusable conflict detection and scheduling algorithm [Terry et al. 1995].

Bayou schedules updates tentatively by order of issuance. It does not attempt to detect conflicts syntactically. An operation in Bayou has three main components: a database query called the *dependency check*, a database update, and a code fragment called the *merge procedure*. All three are provided by the application running on top of Bayou. The dependency check is in charge of detecting conflicts semantically. Typically, it tests if the current replica contents are in a state that would have been acceptable to the user at the time she requested the update. For instance, if the update debits from a bank account, the dependency check might verify that the balance is over a user-defined threshold, even though the balance might have changed since the update was issued. If the dependency check succeeds, Bayou runs the update. If it fails, the merge procedure runs instead. The merge procedure can perform any fix-up necessary. For instance, if the user was trying to set up an appointment, and the dependency check finds that the slot is not free any more, the merge procedure might take a different slot.

Writing a merge procedure in Bayou is not easy. All the restrictions imposed on a state-based resolver (Section 5.2.3) still apply here. Practical experience with Bayou has shown that writing merge procedures for all but the simplest of cases is extremely difficult [Terry et al. 2000].

7.5 Committing Operations

This section examines the protocols for committing updates, i.e., for replicas to agree on an equivalent schedule prefix, allowing them to be finalized. Commitment is used for three purposes. First, a system’s scheduling policy may make non-deterministic choices, in which case the sites must agree on the choice to be applied finally. Second, commitment can inform the user that a particular opera-

tion is permanently applied on all replicas. Third, commitment acts as a space-bounding mechanism, because committed operations can safely be removed from the multi-logs.

7.5.1 Ack Vectors. Acknowledgment vectors (ack vectors) is used in conjunction with timestamp vectors (Section 6.2) to let each site learn about the progress of other sites [Golding 1992]. Site i stores the ack vector AV_i , an N -element array of timestamps. $AV_i[i]$ is defined to be $\min_{j \in \{1 \dots M\}} TV_i[j]$, i.e., site i has received all updates with timestamps no newer than $AV_i[i]$, regardless of their issuers. Ack vectors are exchanged among sites and updated by taking pairwise maxima, just like TVs. Thus, $AV_i[k]$ represents site i 's conservative estimate of the newest update received by site k . With this definition, all updates with timestamps older than $\min_{j \in \{1 \dots N\}} AV_i[j]$ are guaranteed to have been received by all sites, and they can safely be ordered by their timestamps, committed, and deleted from the multi-log. This algorithm requires loosely synchronized physical clocks (Appendix A.2.1) to be used as timestamps. Otherwise, a site with a very slow timestamp could stall the progress of ack vectors of all other sites.

This algorithm is inferior to the primary commit protocol, which we discuss next. First, it can only be used for syntactic, timestamp-based scheduling. It is also prone to live-locking; a single unresponsive site hampers the progress of ack vectors on all other sites, and this problem worsens as the number of sites increases.

7.5.2 Primary Commit Protocol. Bayou implements a primary-commit protocol [Petersen et al. 1997]. In this protocol, all sites agree on a single primary site for a given object (a Bayou transaction cannot access more than one object). The primary unilaterally and totally orders updates by assigning a monotonically increasing commit sequence number (CSN) to each operation it receives. The primary is free to choose any scheduling policy; Bayou uses syntactic, timestamp-based ordering. The mappings between operations and their CSNs are propagated back to other sites by piggybacking on ordinary updates. A non-primary site receiving the CSNs can commit operations in that order and delete them from the multi-log.

Notice the difference between this protocol and single-master replication. In this protocol, all sites are masters. They can issue, exchange, and tentatively apply updates even when the primary is unreachable, because the primary is responsible only for CSN assignment.

7.5.3 Quorum Commit Protocol. The protocol proposed by Deno [Keleher 1999] is pessimistic but also applicable to optimistic environments, because it makes progress even when not all sites are available. Applying the state-machine approach to replication [Lamport 1978] at each step, it runs a vote to commit an update among those outstanding. It assigns a weight to every $\langle \text{site}, \text{update} \rangle$ pair, with the total weight for a given update being 1.0. Every update circulates among sites epidemically. When a site learns of an update, if it does not conflict locally with any other completed update, the site votes in favor of that update. When a site observes that votes for an update reach a plurality, that update commits locally, and it sends a commit notice to all other sites.

In the general case, the commitment protocol considers a single operation at a time serially. An operation that conflicts with a committed operation is aborted unless they commute. Deno allows the application to declare commutativity among operations. Simulation results suggest that this protocol performs similarly to the classic primary-copy scheme [Keleher 1999].

8. ENSURING QUALITY OF CONTENTS

Optimistic replication algorithms cannot ensure single-copy replica consistency. Uniformity only guarantees replica consistency in some hypothetical quiescent state, or up to some virtual point in the past. With updates arriving possibly out of order, and no hard bound on transmission delay, uniformity itself offers little clue to users regarding the *transient* quality of replica contents.

Some replicated services do fine with such a weak guarantee. Usenet and the Porcupine mail

server are such examples [Spencer and Lawrence 1998; Saito et al. 2000]; replica inconsistency is no worse than potential problems caused by NNTP (for Usenet) and SMTP (for email), such as the delay of delivery and duplicate messages. Many services, however, benefit from stronger forms of quality-of-contents guarantees. This section reviews several approaches for controlling the quality of transient replica contents. Section 8.1.2 discusses techniques for preserving causality for object “read” requests as well as for updates. Section 8.2 discusses algorithms for explicitly bounding the amount of inconsistency among replicas. Section 8.3 reviews probabilistic techniques for ensuring the quality of replica contents.

These techniques are (still) not a panacea, in that they either prohibit object access when replicas are found too old or demand synchronous inter-site communication. In other words, they cannot escape the fundamental trade-off between availability and consistency [Fox and Brewer 1999; Yu and Vahdat 2001].

8.1 Preserving Causal Relationship among Accesses

Ordering updates by causality is a basic feature implemented by any optimistic replication system. While it ensures that the final object contents are what users expect, it cannot control the quality of transient contents. For instance, the password database problem (Section 1.3.2) stems from the lack of quality-of-contents guarantee. A solution to this problem is to let users specify which updates should “happen before” a read request, and delay handling the read request until all preceding updates are received. A review of such techniques follows.

8.1.1 Explicit Causal Dependencies. This approach simply lets the user explicitly specify, for each read request, the set of updates that must be incorporated into the replica contents before the read can proceed [Ladin et al. 1992]. This feature is easily implemented in either state- or operation-transfer systems using version vectors. Here, a replica’s state is represented by a version vector. A request’s dependency is represented as a VV as well, and request processing is delayed until the dependency VV dominates the state VV.

8.1.2 Session Guarantees. A problem with the previous approach is that specifying dependency is not a trivial task for users. *Session guarantees* are an approach for automatically generating the appropriate causal dependencies from a user-chosen combination of the following predefined guarantees [Terry et al. 1994; Goel et al. 1998]:

- “Read your writes” (RYW) guarantees that the contents read from a replica incorporate previous writes by the same user. The stale password problem can be solved by enforcing RYW; the user either waits until the new password arrives or receives a “stale replica” error, both of which are less confusing than just reading the old password.
- “Monotonic reads” (MR) guarantees that two successive reads by the same user return increasingly up-to-date contents. For example, in a replicated email service, a user may retrieve the digest of a mailbox and later read one of the messages in the digest. MR guarantees that the user will not get a “non-existent message” error (except, of course, when the user deletes a message in between the two operations).
- “Writes follow reads” (WFR) guarantees that a write request is accepted only after writes observed by previous read requests by the same user are incorporated in the replica.
- “Monotonic writes” (MW) property guarantees that a write request is accepted only after all write requests made by the same user are incorporated in the replica. For example, consider a source code management system. A library module is modified on one replica, and an application program is modified on another in such a way that it depends on the new library module. MW guarantees that the library is updated before the application.

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 3. Implementation of session guarantees.

These properties are automatically guaranteed when the user accesses only a single site, and the replication algorithm preserves the happens-before ordering among updates.¹⁰ In a roaming environment, they are implemented using a *session*. A session, carried by each user (e.g., in a PDA), records two pieces of information: the set of writes issued by the user (*write-set*) and the set of writes the user has observed through object-read requests (*read-set*). The implementations of session guarantees are summarized in Table 3. For example, to ensure RYW, every update request by the user is appended to her write-set. In addition, before answering a read request, a replica checks whether her write-set is a subset of what has been applied by the replica. If so, the read succeeds; otherwise, the user waits or receives a “stale replica” error. Similarly, to ensure MR, for each read request, the replica ensures that the user’s read-set is a subset of those applied by the replica. After replying the request, it requests adding the site’s updates to the user’s read-set.

8.2 Bounding Inconsistency

Some systems explicitly bound the amount of inconsistency among replicas. There is a large body of work about relaxed two-phase locking algorithms for improving the performance of non-replicated database systems [Kumar and Stonebraker 1988; Kumar and Stonebraker 1990; O’Neil 1986; Pu and Leff 1991; Carter et al. 1998; Pu et al. 1995]. As such, they are inapplicable to mobile or wide-area-distributed environments. Nonetheless, there are several systems that target optimistic replication environments, which we review below.

Real-time guarantees, i.e., bounding propagation delay, are already popular in network services. Systems such as WWW [Fielding et al. 1999], NFS [Stern et al. 2001], DNS [Albitz and Liu 2001], and quasi-copies [Alonso et al. 1990] let an object be cached and remain stale for up to a certain number of seconds. Most of them are single-master, pull-based systems, because they can enforce the guarantee by simple periodic polling.

TACT [Yu and Vahdat 2000a; Yu and Vahdat 2000b; Yu and Vahdat 2001] proposes three types of consistency guarantees: real-time, numerical, and order bounding. In TACT, real-time guarantees have the same semantics as in WWW, but is implemented by on-demand pushing (Section 9.3).

Numerical bounding puts a constraint on the difference between replicas’ numerical value. It is implemented by allocating each master replica a “quota” of updates that can be pooled before being pushed to a specific remote replica. Suppose a user’s bank account is replicated at ten master replicas, under the constraint that the balance read on any replica is within \$50 of the actual balance. Then, each master receives a quota of \$5 ($= 50/10$) for the account. Whenever the balance change reaches \$5 on any master, it must push all updates to all other replicas before accepting any new updates.

Order bounding ensures that a replica’s current tentative value differs from the committed value by no more than X number of updates, where X is the limit chosen by the replica. The nature of this guarantee is different from, and perhaps weaker than, numerical guarantees, which ensures that a replica’s tentative value differs no more than X from the tentative value of any other replica. It is implemented by the replica pulling updates from others when the discrepancy between the tentative and committed values exceeds the limit.

¹⁰ In systems that support both tentative updates and epidemic replication (e.g., Bayou), this is not always the case.

8.3 Probabilistic Techniques for Reducing Inconsistency

The techniques discussed in this section rely on the knowledge of the workloads to reduce the replica's average staleness probabilistically with minimum overhead.

Cho and Garcia-Molina [2000] study policies regarding the frequency and order of page re-fetching for web proxies, based on a simplifying assumption that update intervals follow Poisson distributions. They found 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.

Rowstron, Lawrence, and Bishop [2001] did a similar study using real workloads. They present a tool that learns update patterns from a log of past updates using probabilistic modeling [Bishop 1995]. 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 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 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 freshness. A comparison of their adaptive strategy with connecting at fixed periods shows an average freshness improvement of 14%.

9. SCALING OPTIMISTIC REPLICATION SYSTEMS

This section examines the issues in supporting a large number of replicas in optimistically replicated environments. (Scaling to large objects was discussed previously in Section 5.3.) We explain the challenges of supporting many replicas and review three lines of remedies: a structured communication topology, and proactive update pushing and efficient networking.

9.1 Estimating Conflict Rates

Supporting many replicas raises two issues: increased update conflicts and propagation delay. The issue of update conflicts requires some explanation. Gray, Helland, O'Neil, and Shasha [1996] argue that multi-master optimistic replication systems cannot support a large number of master replicas, because they experience $O(M^2)$ update conflicts, whereas single-master systems would experience only $O(M)$ update conflicts.¹¹ Intuitively, concurrent updates are reconciled in multi-master systems, whereas they are delayed or aborted in single-master systems using techniques such as two-phase locking. Because abortion requires a cycle between the updates' causal dependencies, the rate of abortion is far lower than the rate of conflict ($\text{Prob}(\text{abortion}) \approx \text{Prob}(\text{conflict})^2$). Gray et al. make two important assumptions to derive their conclusion: data items are updated equiprobably by all sites, and sites exchange updates with uniform-randomly chosen sites. These assumptions do not hold universally. For example, write-sharing in file systems has been found to be extremely rare [Ousterhout et al. 1985; Baker et al. 1991; Vogels 1999]; the more widely a file is shared, the less likely it is to be written [Wang et al. 2001]. Thus, it is still unclear how severe this conflict problem is in practice.

Both problems — update conflicts and propagation delay — can be alleviated by (1) structuring the communication topology so that any update is propagated to all sites with a small number of “hops”, (2) pushing updates pro-actively as soon as they are issued, or (3) using an efficient network protocol to propagate updates quickly. We review these lines of techniques in the following sections.

9.2 Scaling by Controlling Communication Topology

The perceived rate of conflicts can be reduced by connecting replicas in specific ways. Some examples are shown in Figure 7. The conflict rate is minimized by connecting sites in a star shape,

¹¹ We assume that in the single-master system, the master receives remote update requests from M' (non-master) replicas, and that $M' = M$.

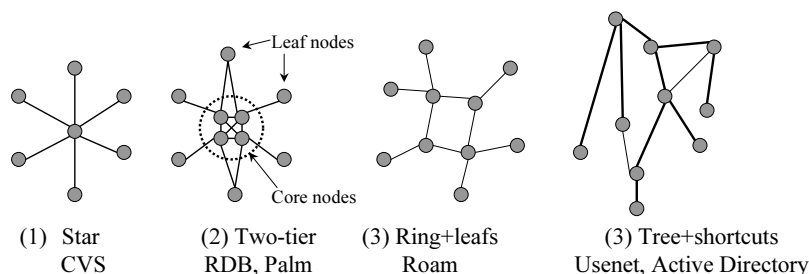


Fig. 7. Examples of replica communication topologies.

because the central site can resolve conflicting updates before they are propagated to peripheral replicas [Wang et al. 2001; Ratner 1998].¹² The star topology also disseminates updates more quickly, because any update reaches any site within two “hops.” CVS is a well-known example (Section 2.3). *Two-tier replication* is a generalized star topology [Gray et al. 1996; Kumar and Satyanarayanan 1993; Rhodes and McKeehan 1998]. Here, sites are split into mostly connected “core nodes” and more weakly-connected “mobile nodes”. The core sites often use a pessimistic replication algorithm to remain consistent with each other, but a leaf uses optimistic replication and communicates only with the core. Note that both the star and two-tier topologies still need to solve all of the problems of multi-master optimistic replication systems — e.g., reliable update propagation, scheduling, and conflict resolution — but they scale better at the cost of sacrificing the flexibility of communication. In contrast, a topology in which each site interacts with a randomly chosen peer (as Gray, Helland, O’Neil, and Shasha [1996] assume) has the slowest propagation speed and the highest conflict rate.

These solutions are not universal. First, there is a trade-off among dissemination speed, load balance, and availability. The star topology puts an inordinate load on the central node, which may in practice retard update propagation. Its central node also becomes the single point of failure. A random topology, on the other hand, is highly available and the load is evenly balanced among sites. Second, sites cannot always choose peers to talk to, especially in mobile ad-hoc networking environment, where communication topology is determined by how humans meet and connect devices.

Several other topologies are used in real-world systems. Many choose a tree topology [Chankhunthod et al. 1996; Yin et al. 1999; Adly 1995; Johnson and Jeong 1996], which mixes the properties of both the star and random topologies. Roam [Ratner 1998] connects core replicas in a ring and hangs other replicas off them. Multi-master systems such as Usenet [Spencer and Lawrence 1998] and Active Directory [Microsoft 2000] can push this idea further and connect sites in a ring or tree structure supplemented by short-cut paths to improve availability and accelerate update propagation.

9.3 Push-Transfer Techniques

In the discussions so far, we have assumed that sites could somehow figure out when they should start transferring updates and to whom. This is a non-issue in some applications. For example, some services can rely on explicit manual synchronization (e.g., PDA), and some that manage few objects can rely on periodic polling (e.g., Web mirroring). In many other environments, however, it is better to *push* updates — i.e., to make a site with an active update responsible for delivering it to others — to reduce the update propagation delay and eliminate the polling overhead. The challenge with pushing is that it may send duplicate updates and increase the network bandwidth consumption and update processing overhead. We start with the simplest pushing scheme, flooding, and then review techniques for reducing its overhead in later sections.

¹² Note the difference with a single-master system. A single master serializes all writes; a star-shaped multi-master system resolves conflicts centrally.

```

—— Global persistent data structures on each site ——
const  $M$  = Number of master sites.
       $N$  = Number of sites ( $N \geq M$ ).
var multiLog: Set(Update) // The set of updates the site has received.
      tm: array [ $0 \dots N-1$ ][ $0 \dots M-1$ ] of Timestamp // The site's timestamp matrix.

m—— Sender side: send updates to dest ——
proc SendUpdate(dest)
  upd  $\leftarrow \emptyset$ 
  for  $0 \leq i < M$ 
    if tm[dest][i] < tm[myself][i] then
      upd  $\leftarrow \text{upd} \cup \{ u \in \text{multiLog} \mid u.\text{issuer} = i \text{ and } u.\text{ts} > \text{tm}[\text{dest}][i] \}$ 
  Send upd and tm to dest.
  tm[dest]  $\leftarrow \text{PairWiseMax}(\text{tm}[\text{myself}], \text{tm}[\text{dest}])$ 

—— Receiver side: called via SendUpdate ——
proc ReceiveUpdate(upd, tmsrc)
  for u  $\in$  upd
    if tm[myself][u.issuer] < u.timestamp then
      multiLog  $\leftarrow \text{multiLog} \cup \{ u \}$ 
      Apply u to the site
  for  $0 \leq i < N, 0 \leq j < M$ 
    tm[i][j]  $\leftarrow \max(\text{tm}[\text{myself}][i][j], \text{tm}_{\text{src}}[i][j])$ 

```

Fig. 8. Site reconciliation using timestamp matrices.

9.3.1 Blind Flooding. Flooding is the simplest pushing scheme. Here, a site with a new update blindly forwards the update to all sites that it has links to. The receiving site uses Thomas's write rule or timestamp vectors to filter out duplicates. This technique is used in Usenet [Lidl et al. 1994; Spencer and Lawrence 1998], NIS [Sun Microsystems 1998], Porcupine [Saito and Levy 2000], and many relational database systems [Oracle 1996].

While being simple, 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 update before sending. Such information is ultimately unavailable in multi-master systems, but several techniques are developed nonetheless, which we overview next.

9.3.2 Link-state Monitoring Techniques. Rumor mongering [Demers et al. 1987] and directional gossiping [Lin and Marzullo 1999] are both probabilistic techniques for suppressing duplicate updates. Rumor mongering starts as a flooding, but each site monitors the number of duplicate updates it has received. It stops forwarding updates when the number of duplicates exceeds a limit. In directional gossiping, each site monitors the number of distinct "paths" updates have traversed. An inter-site link not shared by many paths is likely to be more important, because it may be the sole link that connects this site to another. Thus, the site sends updates 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 update via different paths.

Both techniques are probabilistic in that they sometimes fail to propagate updates to replicas. Thus, for a reliable propagation, the system occasionally needs to resort to plain flooding to flush updates that have been omitted at some sites. Simulation results, however, show that a reasonable parameter setting can nearly eliminate duplicate updates while keeping the reliability of update propagation very close to 100%.

9.3.3 Timestamp Matrices: Estimating the State of Remote Sites. Another technique for reducing duplicate updates is to let each site estimate the progress of others and push only those updates that are likely to be missing at a remote site. *Timestamp matrices*, used for multi-master operation transfer, are one example of such techniques [Wuu and Bernstein 1984; Agrawal et al. 1997].

Site *i* stores a timestamp matrix TM_i , an $N \times M$ matrix of timestamps. $TM_i[i]$ holds *i*'s timestamp vector. The other rows of TM_i hold site *i*'s conservative estimate of the timestamp vectors of other sites. Thus, if $TM_i[k][j] = c$, then site *i* knows that site *k* has received updates issued at site *j* with

timestamps at least up to c . The update propagation procedure, shown in Figure 8, is similar to the one using timestamp vectors (Figure 6). The only difference is that when sending updates to site j , site i uses $TM_i[j]$ as an estimate of site j 's timestamp vector, rather than receiving the vector from j .

9.4 Combining Multiple Protocols

Another class of techniques employs unreliable multicast protocols (e.g., MBONE, Digital fountain [Byers et al. 1998], or the techniques described in Section 9.3.2) to propagate updates efficiently in the common case and to back it up with a slower but reliable algorithm [Lidl et al. 1994; Demers et al. 1987].

9.5 Adding and Removing Masters

A TV contains one entry per master. Thus, its space complexity increases linearly as the system scales. It is more convenient to identify a master by some identifier such as its IP address, rather than by a number between 1 and M ; thus TVs are represented sparsely, as a map of site identifiers to rows. Adding a new master is relatively painless.

Deleting a master that leaves the system permanently is more complex. When an entry for a master Y is missing in another site Z 's TV, Z must determine whether it is because Y is added (which Z still hasn't learned), or because Y is deleted (and Z has forgotten about). Not distinguishing between these cases would cause the system to fetch again updates that had already been applied. Thus, all sites must remove the TV entry for a removed master consistently. Some protocols for doing this are described in [Ratner 1998; Petersen et al. 1997; Adya and Liskov 1997].

Systems that support many replicas are likely to experience frequent replica addition and removal. Many systems indeed require off-line intervention to add or remove sites (e.g., NIS [Sun Microsystems 1998], and Usenet [Saito et al. 1998]), but such a design is feasible only when the site set is mostly static.

Adding and removing sites are trivial in single-master or in pulling state-transfer systems. In fact, this is why services such as Web and FTP mirroring [Nakagawa 1996; Krasel 2000] use pull, state-transfer; they want to install a site, possibly in a different administrative domain without a hassle.

In systems that use a TV or TM, the previous sections assumed that the number of sites (or masters) is precisely known, and that they are numbered consecutively. However, sites join the system and fail dynamically; assigning a dense numbering is impractical. Actual implementations therefore use a sparse representation mapping unique node identifiers (such as static IP addresses) to indices. Adding a new node is easy: if a message is received that contains the identifier of a node not previously known, it is simply added to the vector or matrix of interest, with an assumed previous timestamp value of 0.

Node removal is a problem however. When an entry for a site Y is missing in another site Z 's TV, Z must determine whether it is because Y is added (which Z still hasn't learned), or because Y is deleted (and Z has forgotten about it since). Not distinguishing between these cases would cause the system to incorrectly assume a previous timestamp value of 0, and to fetch a second time updates that had already been applied.

The solution used in Bayou is the following [Petersen et al. 1997]. Consider some site X adding a new site Y to the system. Y 's site identifier is a tuple containing X 's identifier and X 's timestamp vector at the time of creation.¹³ The site creation event is logged and transmitted to other sites. When some other site Z receives a message containing Y 's identifier, and does not currently have for Y in its TV, it can distinguish the two cases:

—If the timestamp embedded in Y 's name is less than $TV_Z[X]$, then Z has not yet received notice of the creation of Y . Z adds Y to its TV, assuming a previous value of 0.

¹³ A site identifier can be arbitrarily long in Bayou.

	Single/State or Op.	Multi/State	Multi/Op.
Availability	low: master is the 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: multi-log
Network overhead	high: $O(\text{object})$		low: $O(\#updates)$

Table 4. Comparing estimated costs of optimistic replication strategies. (Single = single master systems; Multi = multi-master systems; State: state-transfer systems; Op. = operation-transfer systems.)

—Otherwise, Z had previously received notice of the creation of Y , and the only reason it is not currently in the TV is because site Y has been deleted.

10. CONCLUSIONS

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

10.1 Comparing Optimistic Replication Strategies

Table 4 summarizes how different classes of optimistic replication systems introduced in Section 1.4 achieve the high level goals. This table confirms that there is no single winner; each strategy has advantages and disadvantages. Single-master transfer is a good choice if the workload is read-dominated or there is a single writer, because of its simplicity and conflict-freedom. Multi-master state transfer is reasonably simple, and has a low space overhead (consisting of a single timestamp or version vector per object). Its downside is that it cannot exploit operation semantics during conflict resolution, and that its overhead grows with object size. Thus, it is a good choice when objects are naturally small, the conflict rate is low, and conflicts can be resolved by a simple 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 space overhead (the multi-log).

10.2 Summary of Algorithms

Table 5 summarizes the algorithms used to solve the challenges of optimistic replication introduced in Section 1.3. A cell is colored gray to mean that the combination is irrelevant or not an issue. For example, scheduling and conflict management in single-master systems are solved by local concurrency control, and the space overhead of tombstones in state-transfer systems is far lower than that of logs in operation-transfer systems.

10.3 Summary of Optimistically Replicated Systems

Table 6 summarizes the optimistic replication systems mentioned in this survey and the algorithms they use.

10.4 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 processing does improve networking flexibility and scalability. As such, it is hard to do without it, in data-sharing across slow and unreliable networks, in wide-area collaborative work, and in disconnected computing. However appealing, optimistic replication comes at a cost. One is high algorithmic complexity for ensuring uniformity. Conflicts usually require

	Single/State or Op.	Multi/State	Multi/Op.
Update propagation	Thomas's write rule (§5.1)	Thomas's write rule, version vector (§5.2.2)	Timestamp vector (§6.2)
Scheduling	Local concurrency control	Thomas's write rule, version vector	Syntactic or semantic (§7)
Schedule commitment			Operational transformation (§7.3), ack vector (§7.5.1), primary commit (§7.5.2), voting (§7.5.3)
Conflict detection		Two timestamps, version vector	Syntactic or semantic
Conflict resolution		App. specific (§5.2.3)	App. specific (§7.4)
QoC guarantee	Temporal (§8.2)	Temporal, session (§8.1.2)	Temporal, session, numerical, order
Many replicas	Tree topology	Tree topology, two-tier replication	
Many objects	Flooding (§9.3.1), rumor mongering, directed gossiping (§9.3.2)		Flooding, rumor mongering, directed gossiping, timestamp matrix (§9.3.3)
Large objects	Object decomposition, state/op. hybrid		N/A
Space reclamation	Expiration	Expiration, 2-phase reclamation	Expiration, commit.

Table 5. Summary of algorithms. (A cell is colored gray when the combination is irrelevant or not an issue.)

application-specific resolution, and the lost update problem is ultimately unavoidable. Hence some recommendations:

- (1) *Keep it simple.* Pessimistic replication, with many off-the-shelf solutions, is often a perfectly valid answer when working in a fully connected, reliable networking environment. Otherwise, try single-master replication: it is simple, conflict-free, and scales well in practice. 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) *Make objects independent to avoid false negatives.* Most optimistic replication algorithms can maintain consistency between replicas of an object, but not between objects.
- (3) *Quick propagation is the best way to avoid conflicts.* While connected, propagate often and keep replicas in close synchronization. This will minimize divergence when disconnection does occur.
- (4) *Design objects to be conflict-free whenever possible.* Some hints in this direction follow:
 - Partition large objects into smaller ones to avoid false positives (Section 5.3).
 - Design operations to be commutative (Section 7.2.1). For example, use monotonic data structures such as an append-only log, a monotonically increasing counter, or a union-only set.
- (5) *Use simple, structured topologies* such as the star-shaped or two-tiered topologies. These are easier to control and to scale than a purely peer-to-peer network.

Acknowledgements

We thank the following people for reading drafts of this paper and giving us valuable feedback: Miguel Castro, Yek Chong, Christos Karamanolis, Anne-Marie Kermarec, Dejan Milojicic, Ant Rowstron, Susan Spence, and John Wilkes.

	Update propagation	Scheduling	Conflict resolution	Commitment
WWW/FTP mirroring [Nakagawa 1996]	Single/State, pulling	Single master		
DNS [Mockapetris and Dunlap 1988; Albitz and Liu 2001], NIS [Sun Microsystems 1998]	Single/State, blind push/pull	Single Master		
Clearinghouse [Demers et al. 1987]	Multi/State, push	Thomas's write rule		expiration
Active Directory [Microsoft 2000]	Multi/State, pull	Thomas's write rule		expiration
Usenet [Spencer and Lawrence 1998]	Multi/Op, blind pushing	Thomas's write rule		expiration
Bayou [Terry et al. 1995; Petersen et al. 1997]	Multi/Op, timestamp vector	syntactic	semantic	primary commit
TSAE [Golding 1992]	Multi/Op, timestamp vector	syntactic	none (semantic)	Ack vector
IceCube [Kermarrec et al. 2001; Preguiça and Shapiro 2001]	Multi/Op, timestamp vector	semantic	semantic	(Not documented)
TACT [Yu and Vahdat 2000a; Yu and Vahdat 2001]	Multi/op, timestamp vector	syntactic	semantic	primary commit
Ficus, Roam [Ratner 1998]	Multi/State, pull	version vector		two-phase consensus.
Palm Pilot [Rhodes and McKeehan 1998]	Multi/State, pull	two timestamps		expiration
Deno [Keleher 1999]	Multi/State, pull, timestamp vector	voting		
Wuu&Bernstein [Wuu and Bernstein 1984]	Multi/Op, timestamp matrix	commuting operations		N/A
CVS [Cederqvist et al. 2001]	Multi/Op, pull, star topology	syntactic	syntactic	manual, primary commit

Table 6. Summary of systems.

APPENDIX

A. THE ORDER OF EVENTS IN A DISTRIBUTED SYSTEM

This appendix is a brief reminder of a few important distributed systems concepts and algorithms.

A.1 The Happens-Before Relationship Among Updates

Lamport's "happens-before" relation and causality are central concepts in replica consistency management. Update U_1 *happens before* (or precedes) another update U_2 when either:

- Both the updates originated at the same site, and U_1 was issued before U_2 , or
- U_1 originated at site S_1 , U_2 at a different site S_2 , and U_2 is issued after S_2 received U_1 from S_1 , or
- There exists U_3 such that U_1 *happens before* U_3 and U_3 *happens before* U_2 .

Two updates are *concurrent* when neither happens-before the other. Since neither causally depends on the other, they might conflict.

The happens-before relationship is used in optimistic replication mainly in two ways: update scheduling and conflict detection. For the former, the system must choose a schedule compatible with the happens-before relationship to make the final results intuitive to users. The latter requires a little more; the system needs to detect not only happens-before relationship, but also its absence, i.e., concurrency. Below, we review two classes of techniques for expressing and/or detecting happens-before relationship among updates: scalar clocks, which can express happens-before but cannot detect its absence, and vector clocks, which can do both.

A.2 Scalar Clocks

A scalar clock is a single scalar value that records some form of "newness" of a replica or an update.

A.2.1 Physical Clocks. Physical clocks can capture the happens-before relationship between updates from a single site; this is called the FIFO property [Birman 1993]. If properly synchronized, they capture the same relationship globally. Physical clocks are ubiquitous and easy to synchronize. With a negligible cost, modern algorithms can keep clock skew within a few microseconds in a LAN, and a few milliseconds in a wide-area network [Mills 1994]; this is enough to capture most causality relationships that happen in practice. Another benefit is that they can capture "hidden" causality. For example, if a user contacts two different sites sequentially, the accesses are causally related according to the user, but not according to the formal definition. Physical clocks can capture causality even in such cases.

The downside of physical clocks is that they are inaccurate unless perfectly synchronized. Consider an object with two replicas R_1 and R_2 . Say R_1 issues update U . R_2 receives U and then issues update U' (thus, U "happened before" U'). If R_2 's physical clock lags far behind R_1 's, U' 's timestamp may be smaller than U 's, violating their causality relationship. Because clock synchronization is merely a best-effort service in asynchronous networks [Chandra and Toueg 1996], the above situation cannot ultimately be avoided.

A.2.2 Logical Clocks. A logical clock (or Lamport clock) [Lamport 1978] is a counter maintained on each node. It increments before an update is issued at the node. In addition, every network message is tagged with the sender's logical clock. Upon receiving the message, the receiver sets its logical clock to be the larger of its current value and the value found in the message. With this definition, if an update U_2 causally follows update U_1 , then U_2 's logical clock (i.e., the clock value of the node that issued U_2 , at the moment of issuance) is guaranteed to be larger than U_1 's logical clock. Thus, logical clocks are more accurate than physical clocks.

A.2.3 Counters. Some systems use an integer counter that increments just before a replica is modified [Kawell Jr. et al. 1988]. It cannot capture causality of updates from different sites, but it is useful when one wants updates from active sites to be considered "newer" than those from idle

sites. Other systems concatenate two types of clocks to reap the benefits from both, e.g., physical and logical clocks [Terry et al. 1995] or a counter and a physical clock [Microsoft 2000].

A.2.4 Limitations of Scalar Clocks. Scalar clocks have one limitation: they impose a total ordering on causality relationships, which are naturally a partial ordering. This problem exhibits itself in two ways. First, these clocks cannot *detect* concurrent updates [Babaoglu 1993] — an update with a larger clock value does not necessarily causally follow another with a smaller clock value. Second, when used to express update causality, i.e., to order updates, they introduce artificial dependencies (or false-positive causality). These problems are solved by timestamp vectors as described below.

A.3 Timestamp Vectors

A *timestamp vector*, also known as a vector clock, or a multipart timestamp, is a data structure that succinctly captures the happens-before relationship in a distributed system [Parker et al. 1983; Fidge 1988; Mattern 1989]. This paper describes several uses of timestamp vectors: for instance, for ordering and optimizing update propagation in operation-transfer systems (Section 6.2), and for conflict detection in a state-transfer system (Section 5.2.2). Update ordering uses a single vector, maintained cooperatively at all sites. Conflict detection uses a vector per object, known as its *version vector*.

A timestamp vector TV_i , kept on site i , is an M -element array of timestamps that summarizes the current state of the site. Any of the algorithms described in the previous section can be used as timestamps. $TV_i[i]$ on site i shows the last time an update was issued at site i . Thus, $TV_i[i]$ is also called site i 's timestamp. If $TV_i[j] = t$, this means that site i has received updates issued at site j and with timestamps up to t .

A partial order is defined on TVs. $TV_1 < TV_2$ (or TV_2 “dominates” TV_1) if and only if $\forall k \in \{1 \dots M\}$, $TV_1[k] \leq TV_2[k]$ and $\exists m \in \{1 \dots M\}$, $TV_1[m] < TV_2[m]$. This occurs if and only if all updates applied at TV_1 “happened-before” those applied at TV_2 .

By attaching a site's TV to each update issued by the site, timestamp vectors accurately keep track of the happens-before relation or its absence between updates [Fidge 1988; Mattern 1989]. It has been proved to be the smallest data structure that can accurately capture causality [Charron-Bost 1991]. Two useful generalizations of TVs are version vectors (a TV per object; see Section 5.2.2) and timestamp matrices (an N -sized vector of TVs; see Section 9.3.3).

REFERENCES

- ADLY, N. 1995. *Management of Replicated Data in Large Scale Systems*. Ph. D. thesis, Corpus Cristi College, U. of Cambridge. <http://ftp.cl.cam.ac.uk/opera/naa-thesis.ps.gz>.
- 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, Aug. 1997), pp. 73–82.
- AGRAWAL, D., ABBADI, A. E., AND STEIKE, R. C. 1997. Epidemic algorithms in replicated databases. In *16th Symp. 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, Sebastopol, CA, USA. ISBN 0-596-00158-4, <http://www.oreilly.com/catalog/dns4/index.html>.
- ALMARODE, J. AND BRETLE, R. 1998. Reduced-conflict objects. *Journal of Object-Oriented Programming* 10, 8 (Jan.), 40–44.
- 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 (Sept.), 359–384.
- AMIR, Y. 1995. *Replication using group communication over a partitioned network*. Ph. D. thesis, Hebrew University of Jerusalem. <http://www.cs.jhu.edu/~yairamir/phd.ps.gz>.
- BABAOGU, O. 1993. *Distributed Systems*, Chapter 4, pp. 55–96. Addison-Wesley.
- BAKER, M., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K., AND OUSTERHOUT, J. K. 1991. Measurements of a distributed file system. In *13rd Symp. on Op. Sys. Principles (SOSP)* (Pacific Grove, CA, USA, Oct. 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)* (Oct. 1998). ACM/IEEE. <http://www.cis.upenn.edu/~bcpierce/papers/snc-mobicom.ps>.

- BERGER, M., SCHILL, A., AND VÖLKSEN, G. 1996. Supporting autonomous work and reintegration in collaborative systems. In W. CONEN AND G. NEUMANN Eds., *Int. W. on Coord. Techn. for Collaborative Apps.* (Singapore, Dec. 1996), pp. 177–198. Springer. <http://www.inf.tu-dresden.de/lsrn/Ps-Files/autonomous.ps>.
- BERNSTEIN, P. AND NEWCOMER, E. 1997. *Principles of Transaction Processing*. Morgan Kaufmann.
- BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison Wesley.
- BIRMAN, K. 1993. The process group approach to reliable distributed computing. *Commun. ACM* 36, 12 (Dec.), 37–53.
- BIRMAN, K. AND JOSEPH, T. 1987. Reliable communication in the presence of failures. *ACM Trans. on Comp. Sys. (TOCS)* 5, 1 (Feb.), 272–314.
- BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. on Comp. Sys. (TOCS)* 9, 3 (Aug.), 47–76.
- BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. D. 1982. Grapevine: An exercise in distributed computing. *Commun. ACM* 25, 4 (Feb.), 260–274.
- BISHOP, C. M. 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.
- BLAUSTEIN, B. T. AND KAUFMAN, C. W. 1985. Updating replicated data during communication failures. In *11th Int. Conf. on Very Large Data Bases (VLDB)* (Stockholm, Sweden, Aug. 1985), pp. 49–58.
- BOLINGER, D. AND BRONSON, T. 1995. *Applying RCS and SCCS*. O'Reilly & Associates, Sebastopol, CA, USA.
- BYERS, J. W., LUBY, M., MITZENMACHER, M., AND REGE, A. 1998. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM* (Vancouver, BC, Canada, Aug. 1998), pp. 56–67. <http://www.eecs.harvard.edu/~michaelm/NEWWORK/postscripts/multicast-conf.ps.gz>.
- 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)* (March 2001). <http://www.cs.utexas.edu/users/dahlin/papers/failure.ps>.
- CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1996. The weakest failure detector for solving consensus. *Journal of the ACM* 43, 4 (July), 685–722. <http://www.cs.cornell.edu/home/sam/FDpapers/CHT96-JACM.ps>.
- CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43, 2 (March), 225–267. <http://www.cs.cornell.edu/home/sam/FDpapers/CT96-JACM.ps>.
- 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.* (Jan. 1996), pp. 153–164. <ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/HarvestCache.ps.Z>.
- CHARRON-BOST, B. 1991. Concerning the size of logical clocks in distributed systems. *Information Processing Letters* 39, 1 (July), 11–16.
- CHO, J. AND GARCIA-MOLINA, H. 2000. Synchronizing a database to improve freshness. In *Int. Conf. on Management of Data* (Dallas, TX, USA, May 2000), pp. 117–128. <http://www-db.stanford.edu/~cho/papers/cho-synch.pdf>.
- CORMACK, G. V. 1995. A calculus for concurrent update (abstract). In *14th Symp. on Princ. of Distr. Comp. (PODC)* (Ottawa, Canada, Aug. 1995), pp. 269.
- COX, L. P. AND NOBLE, B. D. 2001. Fast reconciliations in fluid replication. In *Int. Conf. on Dist. Comp. Sys. (ICDCS)* (April 2001). <http://mobility.eecs.umich.edu/papers/icdcs01.pdf>.
- DATE, C. J. 2000. *An Introduction to Database Systems* (7th ed.). Addison-Wesley.
- DEDIEU, O. AND PACITTI, E. 2001. Optimistic replication for collaborative applications on the Web. In *Int. W. on Info. Integration on the Web – Techno. and App. (WIIW)* (Rio de Janeiro, Brazil, April 2001), pp. 179–183. [http://www.cos.ufrj.br/wiiw/papers/24-Olivier.Dedieu\(02\).pdf](http://www.cos.ufrj.br/wiiw/papers/24-Olivier.Dedieu(02).pdf).
- 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, Aug. 1987), pp. 1–12.
- DIETTERICH, D. J. 1994. DEC data distributor: for data replication and data warehousing. In *Int. Conf. on Management of Data* (Minneapolis, MN, USA, May 1994), pp. 468. ACM.
- FEKETE, A., GUPTA, D., LUCHANGCO, V., LYNCH, N., AND SHVARTSMAN, A. 1996. Eventually serializable data services. In *15th Symp. on Princ. of Distr. Comp. (PODC)* (Philadelphia, PA, USA, May 1996), pp. 300–309.
- FERREIRA, P., SHAPIRO, M., BLONDEL, X., FAMBON, O., GARCIA, J., KLOOSTERMAN, S., RICHER, N., ROBERTS, M., SANDAKLY, F., COULOURIS, G., DOLLIMORE, J., GUEDES, P., HAGIMONT, D., AND KRAKOWIAK, S. 2000. PerDiS: design, implementation, and use of a PERSistent DIstributed Store. In S. KRAKOWIAK AND S. K. SHRIVASTAVA Eds., *Recent Advances in Distributed Systems*, Volume 1752 of *Lecture Notes in Computer Science*, Chapter 18, pp. 427–452. Springer-Verlag. <http://www-sor.inria.fr/publi/PDIUPDS.lncs1752.html>.

- 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://info.internet.isi.edu/in-notes/rfc/files/rfc2616.txt>.
- 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.
- FOX, A. AND BREWER, E. A. 1999. Harvest, yield, and scalable tolerant systems. In *W. on Hot Topics in Op. Sys. (HOTOS)* (Rio Rico, AZ, USA, March 1999), pp. 174–178. <http://www.csd.uch.gr/~markatos/papers/hotos.ps>.
- GIFFORD, D. K. 1979. Weighted voting for replicated data. In *7th Symp. on Op. Sys. Principles (SOSP)* (Pacific Grove, CA, USA, Dec. 1979), pp. 150–162.
- GOEL, A., PU, C., AND POPEK, G. 1998. View consistency for optimistic replication. In *17th Symp. on Reliable Dist. Sys.* (Oct. 1998), pp. 36–42. <http://www.cse.ogi.edu/~ashvin/publications/consistency.ps>.
- 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, <ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-52.ps.Z>.
- GRAY, J., HELLAND, P., O’NEIL, P., AND SHASHA, D. 1996. Dangers of replication and a solution. In *Int. Conf. on Management of Data* (Montréal, Canada, June 1996), pp. 173–182.
- GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann.
- GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. 2000. Scalable, distributed data structures for Internet service construction. In *4th Symp. on Op. Sys. Design and Implemen. (OSDI)* (San Diego, CA, USA, Oct. 2000), pp. 319–332. <http://www.cs.washington.edu/homes/gribble/papers/dds.ps.gz>.
- GUY, R. G., POPEK, G. J., AND THOMAS W. PAGE, J. 1993. Consistency algorithms for optimistic replication. In *Proc. First IEEE Int. Conf. on Network Protocols* (Oct. 1993).
- GWERTZMAN, J. AND SELTZER, M. 1996. World-Wide Web cache consistency. In *USENIX Winter Tech. Conf.* (Feb. 1996), pp. 141–152.
- HADZILACOS, V. AND TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425 (may), Cornell University. <http://cs-tr.cs.cornell.edu/Dienst/UI/1.0/Display/ncstrl.cornell/TR94-1425>.
- HERLIHY, M. 1986. A quorum-consensus replication method for abstract data types. *ACM Trans. on Comp. Sys. (TOCS)* 4, 1 (Feb.), 32–53.
- JOHNSON, T. AND JEONG, K. 1996. Hierarchical matrix timestamps for scalable update propagation. <http://w4.lns-cornell.edu/~jeong/index-directory/hmt-paper.ps>.
- KANTOR, B. AND RAPSEY, P. 1986. RFC977: Network News Transfer Protocol. <http://info.internet.isi.edu/in-notes/rfc/files/rfc977.txt>.
- KAWELL JR., L., BECKHART, S., HALVORSEN, T., OZZIE, R., AND GREIF, I. 1988. Replicated document management in a group communication system. In *2nd ACM Conf. on Computer-Supp. Coop. Work* (Portland, OR, USA, Sept. 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. <http://mojo.cs.umd.edu/papers/podc99.pdf>.
- 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, Aug. 2001). <http://research.microsoft.com/research/camdis/Publis/podc2001.pdf>.
- KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)* 10, 5 (Feb.), 3–25.
- KRASEL, C. 2000. Leafnode: an NNTP server for small sites. <http://www.leafnode.org>.
- KRONENBERG, N. P., LEVY, H. M., AND STRECKER, W. D. 1986. VAXclusters: A closely-coupled distributed system. *ACM Trans. on Comp. Sys. (TOCS)* 4, 4, 130–146.
- KUMAR, A. AND STONEBRAKER, M. 1988. Semantic based transaction management techniques for replicated data. In *ACM SIGMOD Int. Conf. on Management of Data* (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 *Sixth Int. Conf. on Data Eng.* (Los Angeles, CA, USA, 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.* (Jan. 1993), pp. 202–213. <http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/sicpds.pdf>.
- KUMAR, P. AND SATYANARAYANAN, M. 1995. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.* (New Orleans, LA, USA, Jan. 1995), pp. 95–106. <http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/usenix95.pdf>.

- 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. *Commun. ACM* 21, 7 (July), 558–565.
- LIDL, K., OSBORNE, J., AND MALCOLM, J. 1994. Drinking from the firehose: Multicast USENET news. In *USENIX Winter Tech. Conf.* (1994), pp. 33–45. <http://ftp.uu.net/networking/news/muse/usenix-muse.ps.gz>.
- LIN, M. J. AND MARZULLO, K. 1999. Directional gossip: Gossip in a wide-area network. In *Third European Dependable Computing Conference* (Prague, Czech, Sept. 1999), pp. 364–379. A longer version is published as UC San Diego TR CS99-0622, <http://www-cse.ucsd.edu/~marzullo/edcc.ps>.
- 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). <http://www.informatik.tu-darmstadt.de/VS-/Publikationen/>.
- 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, Sept. 1994), pp. 317–327.
- MOCKAPETRIS, P. V. 1987. RFC1035: Domain names — implementation and specification. <http://info.internet.isi.edu/in-notes/rfc/files/rfc1035.txt>.
- MOCKAPETRIS, P. V. AND DUNLAP, K. 1988. Development of the Domain Name System. In *ACM SIGCOMM* (Stanford, CA, USA, Aug. 1988), pp. 123–133.
- MOORE, K. 1995. The Lotus Notes storage system. In *Int. Conf. on Management of Data* (San Jose, CA, USA, May 1995), pp. 427.
- MOSER, L. E., MELLIER-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. 1996. Totem: A fault-tolerant multicast group communication system. *Commun. ACM* 39, 4 (April), 54–63.
- 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, Dec. 1995), pp. 143–155.
- 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.
- 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, Dec. 1985), pp. 15–24.
- PALMER, C. AND CORMACK, G. 1998. Operation transforms for a distributed shared spreadsheet. In *Conf. on Comp.-Supported Coop. Work* (Seattle, WA, USA, Nov. 1998), pp. 69–78. <http://www.cs.cmu.edu/~crpalmer/pubs-.html/cscw98.ps.gz>.
- 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 Transactions on Soft. Eng. SE-9*, 3, 240–247.
- PATÍÑO-MARTÍNEZ, M., JÍMENEZ-PERIS, R., KEMME, B., AND ALONSO, G. 2000. Scalable replication in database clusters. In *14th Int. Conf. on Dist. Computing (DISC)* (Toledo, Spain, Oct. 2000), pp. 315–329.
- 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, Oct. 1997), pp. 288–301.
- PREGUIÇA, N. AND SHAPIRO, M. 2001. An efficient, scalable algorithm for history-based reconciliation. Submitted for publication.
- 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 (Jan.), 85–109. <http://ftp.cse.ogi.edu/pub/esr/reports/distr-div-control.ps.gz>.
- PU, C. AND LEFF, A. 1991. Replica control in distributed systems: An asynchronous approach. In *Int. Conf. on Management of Data* (Denver, CO, USA, May 1991), pp. 377–386. <http://ftp.cse.ogi.edu/pub/esr/reports/sigmod-91-.ps.gz>.
- PURDY, G. N. 2000. *CVS Pocket Reference*. O’Reilly & Associates, Sebastopol, CA, USA. <http://www.oreilly.com/catalog/cvspr/>.
- 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. <http://www.research.att.com/~misha/epidemic/EDBT96.ps.gz>.
- RAMSEY, N. AND CSIRMAZ, E. 2001. An algebraic approach to file synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng. (FSE)* (Austria, Sept. 2001). <http://www.eecs.harvard.edu/~nr/pubs/sync-abstract.html>.

- 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, <http://ficus-www.cs.ucla.edu/ficus-members/ratner/papers/diss.ps.gz>.
- RAVIN, E., O'REILLY, T., DOUGHERTY, D., AND TODINO, G. 1996. *Using and Managing UUCP*. O'Reilly & Associates, Sebastopol, CA, USA. <http://www.oreilly.com/catalog/umuucp/>.
- REIHER, P., HEIDEMANN, J. S., RATNER, D., SKINNER, G., AND POPEK, G. J. 1994. Resolving file conflicts in the Ficus file system. In *USENIX Annual Tech. Conf.* (June 1994), pp. 183–195. ftp://ftp.cs.ucla.edu/pub/ficus-/usenix_summer_94_resolver.ps.gz.
- RHODES, N. AND MCKEEHAN, J. 1998. *Palm Programming: The Developer's Guide*. O'Reilly.
- ROWSTRON, A. I. T., LAWRENCE, N., AND BISHOP, C. M. 2001. Probabilistic modelling of replica divergence. In M. SHAPIRO Ed., *W. on Hot Topics in Op. Sys. (HOTOS)* (Schloss Elmau, Elmau, Germany, May 2001). IEEE Computer Tech. Committee on Op. Sys. <http://www.research.microsoft.com/~antr/papers/sync.ps.gz>.
- SAITO, Y., BERSHAD, B. N., AND LEVY, H. M. 2000. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. *ACM Trans. on Comp. Sys. (TOCS)* 18, 3 (Aug.), 298–333.
- SAITO, Y. AND LEVY, H. M. 2000. Optimistic replication for Internet data services. In *14th Int. Conf. on Dist. Computing (DISC)* (Toledo, Spain, Oct. 2000), pp. 297–314. <http://www.cs.washington.edu/homes/yasushi/disc00-.pdf>.
- SAITO, Y., MOGUL, J., AND VERGHESE, B. 1998. A Usenet performance study. <http://www.research.digital.com-/wrl/projects/newsbench/>.
- SPENCER, H. AND LAWRENCE, D. 1998. *Managing Usenet*. O'Reilly & Associates, Sebastopol, CA, USA. <http://www.oreilly.com/catalog/musenet/>.
- STERN, H., EISLEY, M., AND LABIAGA, R. 2001. *Managing NFS and NIS* (2nd ed.). O'Reilly & Associates, Sebastopol, CA, USA. ISBN 1-56592-510-6, <http://www.oreilly.com/catalog/nfs2/>.
- SUN, C. 2000. Undo any operation at any time in group editors. In *Conf. on Comp.-Supported Coop. Work* (Philadelphia, PA, USA, Dec. 2000), pp. 191–200.
- SUN, C. AND ELLIS, C. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Coop. Work* (Seattle, WA, USA, Nov. 1998), pp. 59–68.
- SUN MICROSYSTEMS. 1998. Sun Directory Services 3.1 administration guide. <http://www.sun.com/sims/Docs-4.0-/html/sunds/admin/>.
- 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 *Proceedings Int. Conf. on Parallel and Dist. Information Sys. (PDIS)* (Austin, TX, USA, Sept. 1994), pp. 140–149. <http://www.parc.xerox.com/csl/projects/bayou/pubs/sg-pdis-94/SessionGuaranteesPDIS.ps>.
- 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, Dec. 1995), pp. 172–183.
- THOMAS, R. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS)* 4, 2 (June), 180–209.
- VIDOT, N., CART, M., FERRIÉ, J., AND SULEIMAN, M. 2000. Copies convergence in a distributed real-time collaborative environment. In *Conf. on Comp.-Supported Coop. Work* (Philadelphia, PA, USA, Dec. 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, Dec. 1999), pp. 93–109. <http://www.cs.cornell.edu/vogels/NTFileTraces/index.htm>.
- VOGELS, W., DUMITRIU, D., AGRAWAL, A., CHIA, T., AND GUO, K. 1998. Scalability of the Microsoft cluster service. In *Second USENIX Windows NT Symposium* (Seattle, WA, USA, June 1998). <http://www.cs.cornell.edu-/vogels/clusters/mscs/scalability.htm>.
- VOGELS, W., DUMITRIU, D., BIRMAN, K., GAMACHE, R., MASSA, M., SHORT, R., VERT, J., BARRERA, J., AND GRAY, J. 1998. The design and architecture of the Microsoft cluster service — a practical approach to high-availability and scalability. In *28th Int. Symp. on Fault-Tolerant Computing* (Munich, Germany, June 1998), pp. 422–431. IEEE. <http://www.cs.cornell.edu/vogels/clusters/mscs/ftcs28.htm>.
- 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, Oct. 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://info.internet.isi.edu/in-notes/rfc/-files/rfc2186.txt>.

- WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., KARLIN, A., AND LEVY, H. 1999. On the scale and performance of cooperative Web proxy caching. In *17th Symp. on Op. Sys. Principles (SOSP)* (Kiawah Island, SC, USA, Dec. 1999), pp. 16–31.
- WUU, G. T. J. AND BERNSTEIN, A. J. 1984. Efficient solutions to the replicated log and dictionary problems. In *3rd ACM Symp. on Princ. of Distr. Comp. (PODC)* (Vancouver, Canada, Aug. 1984), pp. 233–242.
- YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1999. Hierarchical cache consistency in a WAN. In *2ndUSENIX Symp. on Internet Tech. and Sys. (USITS)* (Boulder, CO, USA, Oct. 1999), pp. 13–24.
- YU, H. AND VAHDAT, A. 2000a. Design and evaluation of a continuous consistency model for replicated services. In *4th Symp. on Op. Sys. Design and Implemen. (OSDI)* (San Diego, CA, USA, Oct. 2000), pp. 305–318. <http://www.cs.duke.edu/~yhf/osdifinal.ps>.
- YU, H. AND VAHDAT, A. 2000b. Efficient numerical error bounding for replicated network services. In *Int. Conf. on Very Large Databases (VLDB)* (Cairo, Egypt, Sept. 2000), pp. 123–133. <http://www.cs.duke.edu/~yhf/vldbfinal.ps>.
- 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, Oct. 2001), pp. 29–42. <http://www-cse.ucsd.edu/sosp01/papers-/vahdat.pdf>.
- ZHANG, Y., PAXON, V., AND SHENKAR, S. 2000. The stationarity of Internet path properties: routing, loss and throughput. Technical report (May), ACIRI. <http://www.aciri.org/vern/papers/stationarity-May00.ps.gz>.