Handbook of Research on Mobility and Computing:
Evolving Technologies and Ubiquitous Impacts

Maria Manuela Cruz-Cunha
Polytechnic Institute of Cávado e Ave, Portugal

Fernando Moreira
Universidade Portucalense, Portugal

Volume I

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE
Hershey · New York

Senior Editorial Director:	Kristin Klinger
Director of Book Publications:	Julia Mosemann
Editorial Director:	Lindsay Johnston
Acquisitions Editor:	Erika Carter
Development Editor:	Michael Killian
Typesetters:	Michael Brehm, Casey Conapitski, Keith Glazewski, Milan Vrarich Jr. & Deanna Zombro
Production Coordinator:	Jamie Snavely
Cover Design:	Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Copyright © 2011 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher. Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

Library of Congress Cataloging-in-Publication Data

Handbook of research on mobility and computing : evolving technologies and
ubiquitous impacts / Maria Manuela Cruz-Cunha and Fernando Moreira, editors.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-60960-042-6 (hbk.) -- ISBN 978-1-60960-043-3 (ebook) 1. Mobile
computing. 2. Wireless communication systems. I. Cruz-Cunha, Maria Manuela,
1964- II. Moreira, Fernando, 1969 Aug. 16-
QA76.59.H35 2011
004.165--dc22
2010036723

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

Chapter 69
Data Replication Support for Collaboration in Mobile and Ubiquitous Computing Environments

João Barreto
INESC-ID/Technical University Lisbon, Portugal

Paulo Ferreira
INESC-ID/Technical University Lisbon, Portugal

ABSTRACT

In this chapter we address techniques to improve the productivity of collaborative users by supporting highly available data sharing in poorly connected environments such as ubiquitous and mobile computing environments. We focus on optimistic replication, a well known technique to attain such a goal. However, the poor connectivity of such environments and the resource limitations of the equipments used are crucial obstacles to useful and effective optimistic replication. We analyze state-of-the art solutions, discussing their strengths and weaknesses along three main effectiveness dimensions: (i) faster strong consistency, (ii) with less aborted work, while (iii) minimizing both the amount of data exchanged between and stored at replicas; and identify open research issues.

DOI: 10.4018/978-1-60960-042-6.ch069

Copyright © 2011, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

INTRODUCTION

Consider a team of co-workers that wishes to write a report in a collaborative fashion. For such a purpose, they create a document file, replicate it across the (possibly mobile) personal computers each colleague holds, and occasionally synchronize replicas to ensure consistency. Using a text editor, each worker is then able to read and modify her replica, adding her contributions asynchronously with the remaining colleagues. One may envision interesting scenarios of productive collaboration. A user may modify the report even while out of her office, replicated in a laptop or hand-held computer she carries while disconnected from the corporate wired network. Further, such a worker may meet other colleagues carrying their laptops with report replicas and, in an ad-hoc fashion, establish a short term work group to collaboratively work on the report.

Besides the shared document editing application, one may consider a wide range of other applications and systems; examples include asynchronous groupware applications (Wilson, 1991; Carstensen & Schmidt, 1999) such as cooperative engineering or software development (Cederqvist et al, 1993; Fitzpatrick et al., 2004; Chou, 2006), collaborative wikis (Leuf & Cunningham, 2001; Ignat et al., 2007), shared project and time management (Kawell et al., 1988; Byrne, 1999), and distributed file (Nowicki, 1989; Morris et al., 1986) or database systems (Thomas et al., 1990).

The previous example illustrates the potential collaborative scenarios that the emerging environments of ubiquitous and mobile computing (Weser, 1991; Forman & Zahorjan, 1994) allow. One may even conceive more extreme scenarios, where the fixed network infrastructure is even less likely to be present. For example, data acquisition in field work, emergency search-and-rescue operations, or war scenarios (Royer & Chai-Keong, 1999). In all these scenarios, collaboration through data sharing is often crucial to the activities the teams on the field carry out.

Optimistic replication is especially interesting in all previous scenarios, due to their inherent weak connectivity. They are based on mobile networks, whose bandwidth is lower than in local-area wired networks, and where network partitions and intermittent links are frequent. Further, mobile nodes reduce their up-time due to battery limitations. Moreover, a fixed network infrastructure, such as a corporate local area network, may not always compensate for the limitations of mobile networks. In fact, access to such an infrastructure is often supported by wireless connections such as IEEE 802.11 (IEEE, 1997), UMTS (3GP) or GPRS (Ericsson AB, 1998); these are typically costly in terms of price and battery consumption, and are seldom poor or intermittent due to low signal. Hence, access to the fixed infrastructure is often minimized to occasional connections. Furthermore, access to the fixed network infrastructure is often established along a path on a wide-area network, such as the Internet; these remain slow and unreliable (Zhang, Paxson & Shenker, 2000; Dahlin et al., 2003).

Optimistic replication, in contrast to traditional replication (i.e., pessimistic replication), enables access to a replica without a priori synchronization with the other replicas. Hence, it offers highly available access to replicas in spite of the above limitations (among other advantages over traditional replication (Saito & Shapiro, 2005). As collaboration in ubiquitous and mobile computing environments becomes popular, the importance of optimistic replication increases.

Inevitably, however, consistency in optimistic replication is challenging (Fox & Brewer, 1999; Yu & Vahdat, 2001; Pedone, 2001). Since one may update a replica at any time and under any circumstance, the work of some user or application at some replica may conflict with concurrent work at other replicas. Hence, consistency is not immediately ensured. A replication protocol is responsible for disseminating updates among replicas and eventually scheduling them consistently at each of them, according to some consistency criterion. The last step of such a process is called update commitment. Possibly, it may involve rolling back or aborting updates, in order to resolve conflicts.

The usefulness of optimistic replication depends strongly on the effectiveness of update commitment along two dimensions:

- Firstly, update commitment should arrive rapidly, in order to reduce the frequency by which users and application are forced to access possibly inconsistent data.
- Secondly, update commitment should be achieved with a minimal number of aborted updates, so as to minimize lost work.

Orthogonally, such requirements must be accomplished even in spite of the limitations of the very environments that motivate optimistic replication. Namely:

- The relatively low bandwidth of mobile network links, along with the constrained energy of mobile devices, requires an efficient usage of the network.
- Similarly, bandwidth of wired wide-area networks is a limited resource; hence, when such networks are accessible, their bandwidth must be used wisely by the replication protocol.
- Further, node failures and network partitions have non-negligible probability, both in wired wide-area networks and, especially, in mobile networks; their implications are crucial to any distributed system.
- Finally, the memory resources of mobile devices are clearly constrained when compared to desktop computers; the storage overhead of optimistic replication must be low enough to cope with such limitations.

Although much research has focused on optimistic replication, existing solutions fail to acceptably fulfill the above two requirements (rapid update commitment and abort minimization). As we explain later, proposed optimistic replication solutions either do not support update commitment, or impose long update commitment delays in the presence of node failures, poor connectivity, or network partitions. Some commitment approaches are oblivious to any application semantics that may be available; hence, they adopt a conservative update commitment approach that aborts more updates than necessary. Alternatively, semantic-aware update commitment is a complex problem, for which semantically-rich fault-tolerant solutions have not yet been proposed. Finally, more intricate commitment protocols that aim at fulfilling the above requirements have significant network and memory overheads, unacceptable for environments where such resources are scarce.

BACKGROUND

A replicated system maintains replicas[1] of a set of logical objects at distributed machines, called sites (also called replica managers or servers). A logical object can, for instance, be a database item, a file or a Java object. Most replicated systems intend to support some collaborative task that a group of users carries on (Saito & Shapiro, 2005; Davidson, Garcia-Molina & Skeen, 1985).

It is the ultimate objective of a replicated system to ensure that distributed replicas are consistent, according to some criterion that the users of the system expect. Traditional replication employs pessimistic strategies to achieve that objective by reducing the availability of the replicated system (Davidson, Garcia-Molina & Skeen, 1985; Wiesmann et al. 2000).

Each site makes worst-case assumptions about the state of the remaining replicas. Therefore its operation follows the premise that, if any inconsistency can occur in result of some replica operation, that operation will not be performed. As a result, pessimistic strategies yield strong consistency guarantees such as linearizability

(Dollimore, Coulouris & Kindberg, 2001) or sequential consistency (Lamport, 1979).

Before accepting an operation request at some replica, pessimistic replication runs some synchronous coordination protocol with the remaining system, in order to ensure that executing the operation will not violate the strong consistency guarantees. For instance, if the request is for a write operation, the system must ensure that no other replica is currently being read or written; possibly, this will mean waiting for other replicas to complete their current operations. In the case of a read operation, the system must guarantee that the local replica has a consistent value; this may imply obtaining the most recent value from other replicas and, again, waiting for other replicas to complete ongoing operations.

In both situations, after issuing a local operation request, a client has to wait for coordination with other remote sites obtaining a response. Such a performance penalty is the first disadvantage of pessimistic replication. Further, if a network partition or a failure of some site should prohibit or halt the coordination protocol, then the request response will as well be disrupted. As a second shortcoming, pessimistic replication offers reduced availability if the latter occur with non-negligible frequency.

Thirdly, pessimistic replication inherently implies that two distinct write operations cannot be accepted in parallel by disjoint sets of sites. It is easy to show that, otherwise, we would no longer be able to ensure strong consistency guarantees. This poses an important obstacle to the scalability of pessimistic systems, as their sites cannot serve operations independently.

Some authors (Saito & Shapiro, 2005) point out a fourth limitation, that some human activities are not adequate to strong consistency guarantees. Rather, such activities are most appropriate to optimistic data sharing. According to such authors, cooperative engineering and code development are examples of such tasks, where users prefer to have continuous access shared data, even when other users are updating it, and possibly generating conflicts.

In the network environments that the present chapter considers, the availability and performance shortcomings clearly constitute strong reasons to not adopt pessimistic replication as our solution. Further, the fourth limitation is also very relevant for our objective of supporting collaborative activities, such as the ones we mention above.

OPTIMISTIC REPLICATION

Optimistic replication (OR), in contrast to pessimistic replication, enables access to a replica without a priori synchronization with the other replicas. Access requests can thus be served just as long as any single site's services are accessible. Write requests that each local replica accepts are then propagated in background to the remaining replicas, and an a posteriori coordination protocol eventually resolves occasional conflicts between divergent replicas.

OR eliminates the main shortcomings of pessimistic replication. Availability and access performance are higher, as the replicated system no longer waits for coordination before accepting a request; instead, applications have their requests quickly accepted, even if connectivity to the remaining replicas is temporarily absent. Also, OR scales to large numbers of replicas, as less coordination is needed and it may run asynchronously in background. Finally, OR inherently supports asynchronous collaboration practices, where users work autonomously on shared contents and only occasionally synchronize.

Inevitably, OR cannot escape the trade-off between consistency and availability. OR pays the cost of high availability with weak consistency. Even if only rarely, optimistic replicas may easily diverge, pushing the system to a state that is no longer strongly consistent (e.g. according to the criteria of linearizability or sequential consistency) and, possibly, has different values that are semantically conflicting (we will define semantic scheduling and conflicts later). Hence, OR can only enforce weak consistency guarantees as replicas unilaterally accept tentative requests.

OR exists behind most Internet services and applications (Saito & Shapiro, 2005), which must cope with the limitations of a wide-area network. Examples include name and directory services, such as DNS (Mockapetris & Dunlap, 1995), Grapevine (Birrell et al., 1982), Clearinghouse (Demers et al., 1987), and Active Directory (Microsoft, 2000); caching, such as WWW caching (Chankhunthod et al., 1996; Wessels &. Claffy, 1997; Fielding et al., 1999); or information exchange services such as Usenet (Lidl, Osborne & Malcolm, 1994) or electronic mail (Saito, Bershad & Levy, 1999). The semantics of these services already assume, by nature, weakly consistent data. In fact, the definition of their semantics was molded a priori by the limitations of wide-area networking; namely, slow and unreliable communication between sites.

Other OR solutions support more generic collaborative applications and semantics, the focus of this chapter. Generic replication systems must cope with a wide semantic space, and be able to satisfy differing consistency requirements. In particular, they must accommodate applications whose semantics are already historically defined in the context of earlier centralized systems or pessimistically replicated systems in local area environments. Here, the OR system should provide semantics that are as close as possible to the original ones, corresponding to users' expectations. This takes us to the notion of eventual consistency.

Eventual Consistency

This chapter focuses on OR systems that attempt to offer *eventual consistency* (EC) (Saito & Shapiro, 2005). EC can be seen as a hybrid that combines weak consistency guarantees (e.g., as those that relaxed consistency Internet services such as USENET or DNS offer) and strong consistency guarantees (as in pessimistic replication systems). At its base, EC offers weak consistency, according to one of the previous criteria. However, EC also ensures that, eventually, the system will agree on and converge to a state that is strongly consistent. Typically, the criterion for strong consistence is sequential consistency.

In EC, we may thus distinguish two stages in the life cycle of an update. Immediately after being issued, the update is optimistically ordered and applied upon a weakly consistent schedule of other updates. In such a stage, we say that the update resulting from the write operation is *tentative*, and the value that we obtain by executing the updates in the weakly consistent schedule is the current tentative value of the replica. Eventually, by means of some distributed coordination, the tentative update becomes ordered in some schedule that ensures strong consistency with the schedules at the remaining replicas, we call it *stable*; similarly, the value that results from the execution of such an ordering is the replica's current stable value.

Some systems are able to distinguish the portion of an ordering of operations whose corresponding updates are already stable, from the portion that is still tentative. These systems offer explicit eventual consistency (EEC). Systems with EEC are able to expose two views over their replica values: the stable view, offering strong consistency guarantees, and the tentative view, offering weak consistency guarantees. Note that some systems offer EC but not EEC; i.e. they ensure that, eventually, replicas converge to a strongly consistent state, but cannot determine when they have reached that state.

EEC is a particularly interesting combination of weak and strong consistency guarantees. It can easily accommodate different applications with distinct correctness criteria and, consequently, distinct consistency requirements. Applications with stronger correctness criteria can access the stable view of replicated objects. Applications with less demanding correctness criteria can enjoy the higher availability of the tentative view.

Eventual Consistency with Bounded Divergence

While EC ensures strong consistence of the stable view, it allows the weakly consistent view to be arbitrarily divergent. For some applications, however, such independence is not acceptable.

Eventual consistency with bounded divergence (Yu & Vahdat, 2000; Santos, Veiga &Ferreira, 2007) strengthens the weakly consistent guarantees of EC, according to some parameterized bounds. More precisely, it imposes a limit on the divergence that tentative values may have, using the stable values as the reference from which divergence is measured. Divergence bounds can be expressed in several ways; from the number of pending tentative updates, to semantic measures such as the sum of money from some account that is withdrawn by updates that are still tentative.

Operations that do not respect the parameterized bounds cannot be accepted and, hence, are either discarded or delayed. Hence, EC with bounded divergence offers lower availability than EC.

Basic Stages toward Eventual Consistency

They run some distributed agreement protocol that is responsible to eventually resolve conflicts and make replicas converge, pushing the system back to a strongly consistent state. Eventual consistency is the central challenge of OR. It may take considerable time, especially when system connectivity is low; it may only be possible at the expense of aborting some tentative work; and it may require substantial storage and network consumptions.

Saito and Shapiro (2005) identify the following basic stages in OR:

1. **Operation submission**. To access some replica, a local user or application submits an operation request. When studying eventual consistency, we are mainly interested in requests that update the object. Update requests may be expressed in different forms, depending on the particular application interface (API) that the replicated system offers. In general, an update request includes, either explicit or implicitly, some precondition for detecting conflicts, as well as a prescription to update the object in case the precondition is verified (Saito & Shapiro, 2005). Internally, the replicated system represents each request by updates[2]. Each site may execute updates upon the value of its replicas, obtaining new versions of the corresponding object. Additionally, each site stores logs of updates along with each replica. Whether update logging is needed or not depends on numerous design aspects of OR, which we will address in the next sections. First, update logging may enable more efficient, incremental replica synchronization. Second, update logging may be necessary for correctly ensuring eventual consistency. Third, update logging is useful for recovery from user mistakes or system corruption (Santry et al., 1999), backup (Cox & Noble, 2002; Quinlan & Dorward, 2002; Storer et al., 2008), post-intrusion diagnosis, (Strunk et al., 2000), or auditing for compliance with electronic records legislation (Peterson et al., 2007).
2. **Update propagation**. An update propagation protocol exchanges new updates (resulting from the above stage) across distributed replicas. The update propagation protocol is asynchronous with respect to the operation submission step that originated the new updates; potentially, it may occur long after the latter.

Different communication strategies may be followed, from optimized structured topologies (Ratner, Reiher & Popek, 1999) to random pair-wise interactions that occur as pairs of sites become in contact (Demers et al., 1987). We call the latter approach epidemic update propagation. Its flexibility is particularly interesting in weakly connected networks, as it allows a site to opportunistically spread its new updates as other sites intermittently become accessible.

3. **Scheduling**. As a replica learns of new updates, either submitted locally or received from other replicas, the replica must tentatively insert the new updates into its local ordering of updates.
4. **Conflict detection and resolution**. Since replicas may concurrently accept new updates, conflicts can occur. In other words, it can happen that some replica, after receiving concurrent updates, cannot order them together with other local updates in a schedule that satisfies the preconditions of all updates. Therefore, complementarily to the scheduling step, each replica needs to check whether the resulting ordering satisfies the preconditions of its updates. Only then is the replica sound according to the semantics of its users and applications.

If a conflict is found, the system must resolve it. One possible, yet undesirable, solution is to abort a sufficient subset of updates so that the conflict disappears. This solution has the obvious disadvantage of discarding tentative work. Some systems try to not to resort to such a solution, by either trying to reorder updates (Kermarrec et al., 2001), by modifying the conflicting updates so as to make them compatible (Terry et al., 1995; Sun & Ellis, 1998), or by asking for user intervention (Kistler &. Satyanarayanan, 91; Cederqvist et al., 1993; Fitzpatrick et al., 2004).

5. **Commitment**. The update orderings at which replicas arrive are typically non-deterministic, as they depend on non-deterministic aspects such as the order in which updates arrived at each replica. Hence, even once all updates have propagated across the system, replicas can have divergent values. The commitment stage runs a distributed agreement protocol that tries to achieve consensus on a canonical update ordering, with which all replicas will eventually be consistent. We say that, once the value of a replica reaches such a condition, it is no longer tentative and becomes stable. Furthermore, we say that the updates that the replica has executed to produce such a stable value are committed. We should note that the moment where a replica value becomes stable is not necessarily observable. This distinguishes systems that offer eventual consistency (EC) from systems providing explicit eventual consistency (EEC). Systems where commitment is explicit are typically able to offer two views on a replica: a tentative view, which results from the current update ordering, possibly including tentative updates; and a stable one, which is the most recent stable value. The former is accessible with high availability, while the latter is guaranteed to be strongly consistent across the stable views of the remaining replicas.

In the following sections, we analyze a number of fundamental design choices that affect the way OR achieves eventual consistency.

Tracking Happens-Before Relations

Tracking happens-before relationships plays a central role in OR (Schwarz & Mattern, 1994). In different stages of OR we need to determine, given two updates (or the resulting versions), whether one happens-before the other one, or whether they are concurrent. For instance, when two sites replicating a common object get in contact, we are interested in determining which replica version happens-before the other one, or if, otherwise, both hold concurrent values. As a second example, systems that offer strict happens-before ordering of tentative updates need to determine whether two updates are related by happens-before or, instead, are concurrent.

Tracking happens-before relations is a well studied problem, and a number of more space- and time-efficient alternatives have been proposed. All aim at representing version sets (which in turn result from the execution of update sets) in some space-efficient manner that allows fast comparisons of version sets. We address such solutions next.

Logical, Real-Time and Plausible Clocks

Logical clocks (Lamport, 1978) and real-time clocks consist of a scalar timestamp. Each site maintains one such clock, and each update has an associated timestamp. If two updates are not concurrent, then the update with the lowest logical/real-time clock happens-before the other update. However, the converse is not true; i.e., neither solution can detect concurrency.

Logical clocks are maintained as follows. Every time the local site issues an update, it increments its logical clock and assigns such a timestamp to the new update. Every time a site receives an update from another site, the receiver site sets its logical clock to a value that is both higher than the site's current clock and the update's timestamp.

Real-time clocks assume synchronized (real-time) clocks at each site. The timestamp of an update is simply the time at which it was first issued. Real-time clocks have the advantage of capturing happens-before relations that occur outside system's control (Saito & Shapiro, 2005). However, the need for synchronized clocks is a crucial disadvantage, as it is impossible in asynchronous environments (Chandra & Toueg, 1996).

Plausible clocks (Valot, 1993; Torres-Rojas & Ahamad, 99) consist of a class of larger, yet constant-size, timestamps that, similarly to logical and real-time clocks, can deterministically track happens-before relationships but not concurrency. Nevertheless, a plausible clock can detect concurrency with high probability.

Version Vectors

Version vectors, in contrast to the previous solutions, can both express happens-before and concurrency relationships between updates (and versions) (Mattern, 1989; Fidge, 1991). A version vector associates a counter with each site that replicates the object being considered. In order to represent the set of updates that happen-before a given update, we associate a version vector to the update.

A usual implementation of a version vector is by means of an array of integer values, where each site replicating the object is univocally associated with one entry in the array, 0,1,2, ... Alternatively, a version vector can also be an associative map, associating some site identifier (such as its IP address) to an integer counter.

Each replica *a* maintains a version vector, VV_a, which reflects the updates that it has executed locally in order to produce its current value. When a new update is issued at replica *a*, the local site increments the entry in VV_a corresponding to itself ($VV_a[a] = VV_a[a]+1$). Accordingly, the new update is assigned the new version vector. As replica *a* receives a new update from some remote replica, b, the site sets $VV_a[i]$ to $maximum(VV_a[i], VV_b[i])$, where *i* is the site that issued the received update. Provided that updates propagate in FIFO order[3], if $VV_a[b] = m$ (for any replica *b*), it means that replica *a* has received every update that replica *b* issued up to its *m*-th update.

Two version vectors can be compared to assert if there exists a happens-before relationship between the updates (or versions) they identify. Given two version vectors, VV_1 and VV_2, VV_1 dominates VV_2, if and only if the value of each entry in VV_2 is greater or equal than the corresponding entry in VV_1. This means that the update (resp. version) that VV_1 identifies happens-before the update (resp. version) with VV_2. If, otherwise, neither VV_1

dominates VV_2, nor VV_2 dominates VV_1, VV_1 and VV_2 identify concurrent updates (resp. versions).

An important limitation of version vectors is that the set of sites that replicate the object being considered is assumed static (hence, the set of replicas is static). Each site has a pre-assigned fixed position within a version vector. This means that replica creation or retirement is prohibited by this basic version vector approach. Secondly, version vectors are unbounded, as the counters in a version vector may grow indefinitely as updates occur. Finally, version vectors neither scale well to large numbers of replicating sites nor to large numbers of objects.

Version Vector Extensions for Dynamic Site Sets

Some variations of version vectors eliminate their assumption of a static site set replicating an object.

For instance, the Bayou replicated system (Petersen et al., 1996) handles replica creation and retirement as special updates that propagate across the system by the update propagation protocol. The sites that receive such updates dynamically add or remove (respectively) the corresponding entries from their version vectors. However, Bayou's approach requires an intricate site naming scheme in which replica identifier size irreversibly increases as replicas join and retire.

Ratner's dynamic version vectors (Ratner,, 1998) are also able to expand and compress in response to replica creation or deletion. Replica expansion is simple: when a given replica issues its first update, it simply adds a new entry to its affected dynamic version vector(s). However, removing an entry requires agreement by every replica, which means that any single inaccessible replica will halt such a process.

Similarly to dynamic version vectors, version stamps (Almeida, Baquero & Fonte, 2002) express the happens-before relation in systems where replicas can join and retire at any time. Furthermore, version stamps avoid the need for a unique identifier per site. Assigning unique identifiers is difficult when connectivity is poor[4], and sites come and go frequently.

Each replica constructs its version stamps, based on the history of updates and synchronization sessions that the local replica has seen, thus obviating the need for a mapping from each site to a unique identifier.

The expressiveness of version stamps is limited. Whereas version vectors express happens-before relationships between any pair of versions/update (including old versions that have already been overwritten), version stamps can only relate the current versions of each replica. Almeida et al. designate such a set of coexisting versions as a frontier (Almeida, Baquero & Fonte, 2002).

This means that, in OR systems that maintain old updates in logs, one cannot use version stamps to determine whether an old, logged update happens-before some other update, as the former update may not belong to the same frontier as the latter. Version stamps are designed for systems that will only ever require comparing updates/versions that co-exist in some moment.

Bounded Version Vectors

Version vectors are unbounded data structures, as they assume that counters may increase indefinitely. Almeida et al. propose a representation of version vectors that places a bound on their space requirements (Almeida, Almeida & Baquero, 2004). Like version stamps, bounded version vectors can only express happens-before relationships between versions in the same frontier.

Vector Sets

Version Vectors represent the set of versions of a given individual replica. This means that the number of version vectors that a site needs to maintain and propagate grows linearly with the number of objects the site replicates. This is an obvious scalability obstacle, as most interesting systems (e.g. distributed file systems) have large numbers of replicated objects.

Malkhi and Terry have proposed Concise Version Vectors (Malkhi & Terry, 2005), later renamed Vector Sets (VSs) (Malkhi, Novik & Purcell, 2007), to solve the scalability problem of version vectors. VSs represent the set of versions of an arbitrarily large replica set in a single vector, with one counter per site, provided that update propagation sessions always complete without faults. Provided that communication disruptions are reasonably rare, VSs dramatically reduce the storage and communication overhead of version vectors in systems with numerous replicated objects (Malkhi & Terry, 2005).

Like version stamps and bounded version vectors, VSs can only express happens-before relationships among versions in the same frontier.

Scheduling and Commitment

The road to eventual consistency has two main stages: first, individual replicas schedule new updates that they learn of in some way that is safe, i.e. free of conflicts; second, each such tentative schedule is submitted as a candidate for some a distributed commitment protocol, which will, from such an input, agree on a common schedule which will then be imposed to every replica.

In the following sections, we address each stage.

Scheduling and Conflict Handling

We distinguish two approaches for update scheduling: syntactic and semantic. The distinction lies on the information that is available to each replica when it is about to schedule a new update.

In the syntactic approach, no explicit precondition is made available by the application that requests the operation causing the update. In this case, scheduling can only be driven by application-independent information such as the happens-before relation between the updates.

Based on such restricted information, syntactic scheduling tries to avoid schedules that may break users' expectations. More precisely, if update u1 happens-before update u2, then a syntactic schedule will order u1 before u2, as it knows that this is the same order in which the user saw the effects of each update. Note that, if the effects of both updates are commutative, the scheduler could also safely order u2 before u1, as the user would not notice it. However, as such a commutativity relationship is not known to the syntactic scheduler, it must assume the worst case where the updates are non-commutable. Hence, in the absence of richer information, syntactic scheduling is a conservative approach that restricts the set of acceptable schedules.

Concurrent updates, however, are not ordered. We find syntactic schedulers that behave differently in such a case. Some will artificially order concurrent updates, either in some total system-wide order (by real time or by site identifier, e.g. (Terry et al., 1995)) or by reception order, which may vary for each replica (e.g., (Guy et al., 1998, Ratner, Reiher & Popek, 1999)). In either solution, the resulting schedule may no longer satisfy users' expectations, as the concurrent updates may be conflicting according to application semantics and, still, the scheduler decides to execute them.

Other syntactic schedulers opt for scheduling only one of the concurrent updates, and exclude the remaining concurrent updates from the schedule (e.g. (Keleher, 1999)). This approach conservatively avoids executing updates that can be mutually incompatible. However, it has the secondary effect of aborting user work, which is evidently undesirable.

The former syntactic approach ensures happens-before ordering of updates, possibly combined with the total ordering, while the latter syntactic approach provides strict happens-before ordering.

In some cases, rich semantic knowledge is available about the updates. Essentially, such information determines the preconditions to schedule an update. The replicated system may take advantage of such preconditions to try out different schedules, possibly violating happens-before relationships, in order to determine which of them are sound according to the available semantic information.

Some systems maintain pre-defined per-operation commutativity tables (Jagadish, Mumick & Rabinovich 1997; Keleher, 1999). Whenever a replica receives concurrent updates, it may consult the table and check whether both correspond to operations that are commutable; if so, they can be both scheduled, in any order.

In other systems, operation requests carry explicit semantic information about the scheduling preconditions of each update. For instance, Bayou's applications provide dependency checks (Terry, 1995) along with each update. A dependency check is an application-specific conflict detection routine, which compares the current replica state to that expected by the update. The output tells whether the update may be safely appended to the end of such a schedule or not, which is equivalent to compatibility or conflict relations between the update and the schedule.

Even richer semantic information about scheduling preconditions may be available, as in the case of IceCube (Kermarrec, 2001), or the Actions-Constraints Framework (Shapiro et al., 2004). Their approach reifies semantic relations between as constraints between updates. It represents such knowledge as a graph where nodes correspond to updates, and edges consist of semantic constraints of different kinds.

We should mention that scheduling has important consequences in other stages of OR, notably update propagation and commitment. Consequently, although rich semantic knowledge permits higher scheduling flexibility, some systems partially abdicate it for efficiency of other stages of OR. For instance, Bayou imposes that scheduling respect the local issuing order, i.e. two updates issued by the same site must always be scheduled in that (partial) order. Such a restriction enables a simpler update propagation protocol, which can use version vectors to determine the set of updates to propagate (Petersen, 1997). Inevitably, it limits scheduling freedom and, thus, increases the frequency aborts.

Semantic information may also help when no schedule is found that satisfies the preconditions of some update, by telling how the update can be modified so that its effects become safe even when the original precondition fails. In Bayou, updates also carry a merge procedure, a deterministic routine that should be executed instead of the update when the dependency check fails. The merge procedure may inspect the replica value upon which the update is about to be scheduled and react to it. Ultimately, the merge procedure can do nothing, which is equivalent to discarding the update.

Finally, other approaches are specialized to particular semantic domains and, using a relatively complex set of rules, are able to safely schedule updates, detecting and resolving any conflict that is already expected in such a semantic domain.

One cannot, however, directly generalize such approaches to other domains. A first example is the problem of optimistic directory replication in distributed file systems. The possible conflicts and the possible safe resolution actions are well studied, for instance in the algebra proposed by Ramsey and Csirmaz (2001), or in the directory conflict resolution procedures of the Locus (Walker, 1983) and Coda (Kistler & Satyanarayanan, 1991) replicated file systems.

Other work follows the Operational Transformation method (Sun & Ellis, 1998) to ensure consistency in collaborative editors. Instead of aborting updates to resolve conflicts, this method avoids conflicts by transforming the updates, taking advantage of well known semantic rules of the domain of collaborative editing. Operational Transformation solutions are complex. They typically assume only two concurrent users (Shapiro & Saito, 2005) and are restricted to very confined application domains. Research on this method is traditionally disjointed with the work on general-purpose OR that this chapter addresses.

Update Commitment

Commitment is a key aspect of OR with eventual consistency. One may distinguish four main commitment approaches in OR literature.

A first approach may be designated as the unconscious approach (Baldoni et al., 2006). In this case, the protocol ensures eventual consistency; however, applications may not determine whether replicated data results from tentative or committed updates.

These systems are adequate for applications with weak consistency demands. For example, Usenet, DNS, Roam and Draw-Together (Ignat & Norrie, 2006) adopt this approach. The protocol proposed by Baldoni et al. (2006) is another example.

Other approaches, however, allow explicit commitment. A second approach for commitment is to have a replica commit an update as soon as the replica knows that every other replica have received the update (Golding, 1993). This approach has two important drawbacks. First, the unavailability of any single replica stalls the entire commitment process. This is a particularly significant problem in loosely-coupled environments. Second, this approach is very simplified, in the sense that it does not assume update conflicts. Instead, it tries to commit all updates (as executed); no updates are aborted. The TSAE algorithm (Golding, 1993) and the ESDS system (Fekete et al., 1996) follow this approach, though by different means. One may also employ timestamp matrices (Wuu & Bernstein, 1984; Agrawal, El Abbadi & Steinke 1997) for such a purpose.

A third approach is a primary commit protocol (Petersen et al., 1997). It centralizes commitment into a single distinguished primary replica that establishes a commitment total order over the updates it receives. Such an order is then propagated back to the remaining replicas. Primary commit is able to rapidly commit updates, since it suffices for an update to be received by the primary replica to become committed.

However, should the primary replica become unavailable, commitment of updates generated by replicas other than the primary halts. This constitutes an important drawback in loosely-coupled networks.

Primary commit is very flexible in terms of the scheduling methods that may rely on primary commit. Examples of systems that use primary commit include Coda (Kistler & Satyanarayanan, 1991), CVS (Cederqvist et al., 1993), Bayou (Petersen et al., 1997), IceCube (Kermarrec, 2001) and TACT (Yu & Vahdat, 2000). In particular, to the best of our knowledge, existing OR solutions that rely on a rich semantic repertoire (namely, Bayou and IceCube) all use primary commit.

Finally, a fourth approach is by means of voting (Pâris & Long, 1988; Jajodia & Mutchler, 1990; Amir & Wool, 1996). Here, divergent update schedules constitute candidates in an election, while replicas act as voters. Once an update schedule has collected votes from a quorum of voters that guarantee the election of the corresponding candidate, its updates may be committed in the corresponding order. Voting eliminates the single point of failure of primary commit.

In particular, Keleher introduced voting in the context of epidemic commitment protocols (Keleher, 1999); his protocol is used in the Deno (Cetintemel et al., 2003) system. The epidemic nature of the protocol allows updates to commit even when a quorum of replicas is not simultaneously connected. The protocol relies on fixed per-object currencies, where there is a fixed, constant weight that is distributed by all replicas of a particular object. Fixed currencies avoid the need of global membership knowledge at each replica, thus facilitating replica addition and removal, as well as currency redistribution among replicas.

Deno requires one entire election round to be completed in order to commit each single update (Cetintemel et al., 2003). This is acceptable when applications are interested in knowing the commitment outcome of each tentatively issued update before issuing the next, causally related, one. However, collaborative users in loosely-coupled networks will often be interested in issuing sequences of multiple consecutive tentative updates before knowing about their commitment, as the latter may take a long time to arrive. In such situations, the commitment delay imposed by Deno's voting protocol becomes unacceptably higher than that of primary commit (Barreto & Ferreira, 2007).

Holliday et al. (2003) have proposed a closely similar approach. Their algorithm also relies on epidemic quorum systems for commitment in replicate databases. However, they propose epidemic quorum systems that use coteries that exist in traditional (i.e. non epidemic) quorum systems, such as majority. To the best of our knowledge, no study, either theoretical or empirical has ever compared Holliday et al.'s majority-based and Deno's plurality-based approaches. In fact, apart from the above mentioned works, epidemic quorum systems remain a relatively obscure field (Barreto & Ferreira, 2008).

The Version Vector Weighted Voting protocol (VVWV) (Barreto & Ferreira, 2007) employs a commitment algorithm that, while relying on epidemic weighted voting, substantially outperforms the above mentioned epidemic weighted voting solutions by being able to commit multiple tentative updates quicker, in a single election round. Situations of multiple pending tentative updates tend to increase when connectivity is weak, thus increasing VVWV's advantage.

Complementary Adaptation Techniques

Complementarily to the core of OR protocols, as discussed so far, some work focuses on the adaptation of OR to environments with constrained resources such as the ones this chapter considers.

Partial Replication

Partial replication, in contrast to full replication, allows each replica to hold only a subset of the data items comprising the corresponding object. It may take advantage of access locality by replicating only the data items constituting the whole object that are most likely to be accessed from the local replica.

Partial replication is more appropriate than full replication when hosts have constrained memory resources and network bandwidth is scarce. Moreover, it improves scalability, since only a smaller subset of replicas needs to be involved when the system coordinates write operations upon individual data items (of the whole object). Nevertheless, achieving partial replication is a complex problem that has important implications to most phases of OR, from scheduling and conflict detection to update commitment. It is, to the best of our knowledge, far from being solved in OR.

Schiper et al. have formally studied the problem, and proposed algorithms that extend transactional database replication protocols[5] (Schiper, Schmidt & Pedone, 2006), but not OR protocols. More recently, and again in the context of transactional database replication, Sutra and Shapiro have proposed a partial replication algorithm that avoids computing a total order over operations that are not mutually conflicting (Sutra & Shapiro, 2008).

The PRACTI Replication toolkit (Belaramani et al., 2006) provides partial replication of both data and meta-data for generic OR systems with varying consistency requirements and network topologies. However, PRACTI does not support EEC, which strongly restricts the universe of applications for which PRACTI's consistency guarantees are effectively appropriate.

Full replication is a radically less challenging design choice. Not surprisingly, most existing solutions on OR rely on it.

Advanced Synchronization Schemes

Fluid Replication (Cox & Noble, 2001) divides hierarchically-structured objects (file system trees) into smaller sub-objects, identifying the sub-trees (represented by Least Common Ancestors, or LCAs) that are modified with respect to a base version. This technique exploits the temporal and spatial locality of updates in order to improve update propagation in networks that may be temporarily poorly connected. A client exchanges only meta-data updates (which include LCAs) with the server, deferring update propagation to moments of strong connectivity. The exchange of meta-data enables the client to commit updates it generates, even before their actual propagation to the server. Moreover, deferred propagation may be more network-efficient, as redundant or self-canceling updates in the batch of pending updates need not be propagated. As a drawback, Fluid Replication is limited to client-server systems.

Fluid Replication's separation of control from data is not new in OR. In Coda (Kistler & Satyanarayanan, 1991), servers send invalidations to clients holding replicas of an updated object. Update propagation to invalidated replicas is performed subsequently only. The Pangaea file system (Saito et al., 2002) floods small messages containing timestamps of updates (called harbingers) before propagating the latter.

Barreto et al. (2007) describe how to extend generic OR protocols to generate and disseminate lightweight packets containing consistency meta-data only (such as version vectors). Such packets can considerably contribute to reducing commitment delays and aborted updates, while enabling more efficient updates transfer.

Systems such as Bayou (Petersen et al., 1997), Rumor (Guy et al., 1998), or Footloose (Paluska et al., 2003) allow an off-line form of synchronization, called *off-line anti-entropy.* With offline anti-entropy, the interacting replicas need not to be simultaneously accessible across the network to synchronize, enabling alternative means of anti-entropy. These include transportable storage media, as well as mobile and stationary devices, accessible through the network, that are willing to temporarily carry off-line anti-entropy packets.

Off-line anti-entropy has a higher communication overhead than regular anti-entropy. Whereas, in regular anti-entropy, the sender may ask the receiver what is the minimal set of relevant information to send, that is not possible in off-line anti-entropy. Therefore, packets exchanged in off-line anti-entropy should carry enough information so that they are meaningful for every potential receiver; hence, their size may grow significantly.

Efficient Update Propagation

Many interesting and useful replicated systems require transferring large sets of data across a network. Examples include network file systems, content delivery networks, software distribution mirroring systems, distributed backup systems, cooperative groupware systems, and many other state-based replicated systems. Unfortunately, bandwidth and battery remain scarce resources in mobile networks. We now survey techniques that try to make update propagation more efficient.

In some systems, applications provide a precise operation specification along each operation they request from the replicated system. We call such systems as *operation-based* or *operation-transfer* systems. Typically, operation-based requests are more space-efficient than the alternative of state-based requests, which instead contain the value resulting from the application of the requested operation. Systems such as Bayou (Demers et al., 1994) or IceCube (Kermarrec et al., 2001), or operation transformation solutions (Sun & Ellis, 1998) rely on operation-based requests.

The activity shipping approach (Lee, Leung & Satyanarayanan, 1999; Chang, Velayutham & Sivakumar, 2004) tries to extract operation-based requests from systems that are originally state-based. With activity shipping, user operations are logged at a client computer that is modifying a file and, when necessary, propagated to the server computer. The latter then re-executes the operations upon the old file version and verifies whether the regenerated file is identical to the updated file at the client by means of fingerprint comparison. Activity shipping has the fundamental drawbacks of (a) requiring semantic knowledge over the accessed files and (b) imposing the operating environments at the communicating ends to be identical.

Much recent work has proposed *data deduplication* techniques (Trigdell & Mackerras, 1998; Muthitacharoen, Chen & Mazieres, 2001) for efficient state-based update transfer across the network, which may be combined with conventional techniques such as data compression (Lelewer & Hirschberg, 1987) or caching (Levy & Silberschatz, 1990). Data deduplication works by avoiding transferring redundant chunks of data; i.e. data portions of the updates to send that already exist at the receiving site. The receiving site may hence obtain the redundant chunks locally instead of downloading them from the network, and only the remaining, literal chunks need to be transferred.

The prominent approach of compare-by-hash (Trigdell & Mackerras, 1998; Muthitacharoen, Chen & DavidMazieres, 2001; Cox & Noble, 2002; Jain, Dahlin & Tewari, 2005; Bobbarjung, Jagannathan & Dubnicki, 2006; Eshghi et al., 2007) tries to detect such content redundancy by exchanging cryptographic hash values of the chunks to transfer, and comparing them with the hash values of the receiver's contents. Compare-by-hash complicates the data transfer protocol with additional round-trips (i), exchanged meta-data (ii) and hash look-ups (iii). These may not always compensate for the gains in transferred data volume; namely, if redundancy is low or none, or when, aiming for higher precision, one uses finer-granularity chunks (Jain, Dahlin & Tewari, 2005; Muthitacharoen, Chen & DavidMazieres, 2001;, Bobbarjung, Jagannathan & Dubnicki, 2006). Moreover, any known technique for improving the precision and efficiency of compare-by-hash (Jain, Dahlin & Tewari, 2005; Eshghi et al., 2007) increases at least one of items (i) to (iii).

Earlier alternatives to compare-by-hash, such as delta-encoding (Fitzpatrick et al., 2004; Henson & Garzik, 2002; MacDonald, 2000) and cooperative caching (Spring & Wetherall, 2000), are able to less cases of redundancy. However they can attain higher precision when detecting such cases. Recently, the redFS system (Barreto & Ferreira, 2009) proposed hybrid approach that combines techniques from both delta-encoding, cooperative caching and compare-by-hash, thereby borrowing most advantages that distinguish each such alternative.

FUTURE RESEARCH DIRECTIONS

Despite the success of OR in very specific applications, such as DNS, USENET or electronic mail, it is still hardly applicable to general case collaborative applications. When compared to its pessimistic counterpart, OR introduces crucial obstacles to its collaborative users. Namely, the temporal distance that separates tentative write operations from their actual commitment with strong consistency guarantees; the inherent possibility of lost work due to aborts; and the increased storage and network requirements that maintaining large version logs imposes are substantial obstacles. Ironically, the very network environments that call for OR, as a means of dealing with their inherent weak connectivity, tend to substantially amplify all the above obstacles; mostly, due to their weak connectivity and resource constraints.

Therefore, several important issues remain open problems in OR. The road towards decentralized commitment protocols, particularly appropriate to weakly connected environments such as mobile networks, has not yet found a way to encompass semantic-aware protocols. By neglecting application semantics, current decentralized commitment protocols require more messages and abort more than effectively necessary.

Moreover, OR is still largely oblivious of the idiosyncrasies of the mobile environments. More efficient synchronization and update propagation protocols should be devised, departing from the recent research efforts towards partial replication (SchiperRodrigo Schmidt & Pedone, 2006) and efficient update propagation through data deduplication Trigdell & Mackerras, 1998; Muthitacharoen, Chen & DavidMazieres, 2001).

Battery is another resource that should be taken into account by OR protocols. These should adapt their operation in order to accomplish sufficient consistency while, simultaneously, minimizing energy consumption. For instance, by disconnecting network connections that currently connect the mobile node to no replica that is relevant to the activity that the local user is pursuing.

Finally, most existing protocols consider an isolated world that exclusively comprises the set of sites carrying replicas of some objects, neglecting an increasingly dense neighborhood of other devices, unknown *a priori*. OR should also take advantage of such ubiquitous surroundings to find innovative ways to exchange consistency data and meta-data, therefore reducing the impact of weak connectivity.

CONCLUSION

OR is a fundamental technique for supporting collaborative work practices in a fault-tolerant manner in weakly connected network environments. As collaboration through weakly connected networks becomes popular (e.g. by using asynchronous groupware applications, or distributed file or database systems, and collaborative wikis), the importance of this technique increases. Examples of such weakly connected environments range from the Internet to ubiquitous computing and mobile computing environments.

This chapter surveys fundamental aspects form state-of-the-art solutions to OR and identifies open research issues. We focus on the three crucial requirements for most applications and users: rapid update commitment, fewer aborts and adaptation to network and memory constraints. Namely, we address: consistency guarantees and their trade-off against availability; mechanisms for tracking the happens-before relation among updates and versions; approaches for scheduling and commitment; and complementary adaptation mechanisms.

REFERENCES

Agrawal, D. ElAbbadi, A., & Steinke, R. C. (1997). Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–172, New York, NY, USA, 1997. ACM.

Almeida, J. B., Almeida, P. S., & Baquero, C. (2004). Bounded version vectors. In Rachid Guerraoui, (Ed.), *Proceedings of DISC 2004: 18th International Symposium on Distributed Computing*, number 3274 in LNCS, pages 102–116. Springer Verlag.

Almeida, P. S., Baquero, C., & Fonte, F. (2002). Version stamps - decentralized version vectors. In Proc. of the *22nd International Conference on Distributed Computing Systems*.

Amir, Y., & Wool, A. (1996). Evaluating quorum systems over the internet. In *Symposium on Fault-Tolerant Computing*, pages 26–35.

Baldoni, R., Guerraoui, R., Levy, R. R., Quéma, V., & Piergiovanni, S. T. (2006). Unconscious Eventual Consistency with Gossips. In *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*.

Barreto, J., & Ferreira, P. (2007). Version vector weighted voting protocol: efficient and fault-tolerant commitment for weakly connected replicas. *Concurrency and Computation*, *19*(17), 2271–2283. doi:10.1002/cpe.1168

Barreto, J., & Ferreira, P. (2008). The obscure nature of epidemic quorum systems. In *ACM HotMobile 2008: The Ninth Workshop on Mobile Computing Systems and Applications*, Napa Valley, CA, USA. ACM Press.

Barreto, J., & Ferreira, P. (2009). Efficient Locally Trackable Deduplication in Replicated Systems. In ACM/IFIP/USENIX 10th International Middleware Conference, Urbana Champaign, Illinois, USA. ACM Press.

Barreto, J., Ferreira, P., & Shapiro, M. (2007). Exploiting our computational surroundings for better mobile collaboration. In *8th International Conference on Mobile Data Management (MDM 2007)*, pages 110–117. IEEE.

Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., & Zheng, J. (2006). PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Birrell, A., Levin, R., Needham, R. M., & Schroeder, M. D. (1982). Grapevine: An exercise in distributed computing. *Communications of the ACM*, *25*(4), 260–274. doi:10.1145/358468.358487

Bobbarjung, D. R., Jagannathan, S., & Dubnicki, C. (2006). Improving duplicate elimination in storage systems. *Transactions on Storage*, *2*(4), 424–448. doi:10.1145/1210596.1210599

Boulkenafed, M., & Issarny, V. (2003). Adhocfs: Sharing files in wlans. In *Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA.

Byrne, R. (1999). *Building Applications with Microsoft Outlook 2000 Technical Reference*. Redmond, WA, USA: Microsoft Press.

Carstensen, P. H., & Schmidt, K. (1999). Computer supported cooperative work: New challenges to systems design. In Kenji Itoh (E.), *Handbook of Human Factors*, pages 619–636. Asakura Publishing. In Japanese, English Version available from http://www.itu.dk/people/schmidt/publ.html.

Cederqvist, P. & al (1993). *Version management with CVS.* [On-line Manual] http://www.cvshome.org/docs/manual/, as of 03.09.2002.

Cetintemel, U., Keleher, P. J., Bhattacharjee, B., & Franklin, M. J. (2003). Deno: A decentralized, peer-to-peer object replication system for mobile and weakly-connected environments. *IEEE Transactions* on Computer Systems (TOCS), 52.

Cetintemel, U., Keleher, P. J., & Franklin, M. J. (2001). Support for speculative update propagation and mobility in deno. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 509–516.

Chandra, T. D., & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, *43*, 225–267. doi:10.1145/226643.226647

Chang, T.-Y., Velayutham, A., & Sivakumar, R. (2004). Mimic: raw activity shipping for file synchronization in mobile file systems. In Proceedings of the 2nd international conference on Mobile systems, applications, and services, pages 165–176. ACM Press.

Chankhunthod, A., Danzig, P. B., Neerdaels, C., Schwartz, M. F., & Worrell, K. J. (1996). A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164.

Chou, Y. (2006). *Get into the Groove: Solutions for Secure and Dynamic Collaboration*. http://technet.microsoft.com/en-us/magazine/cc160900.aspx.

Cox, L., & Noble, B. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*

Cox, L. P., & Noble, B. D. (2001). Fast Reconciliations in Fluid Replication. In International Conference on Distributed Computing Systems (ICDCS), pages 449–458.

Dahlin, M., Baddepudi, B., Chandra, V., Gao, L., & Nayate, A. (2003). End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, *11*(2), 300–313. doi:10.1109/TNET.2003.810312

Davidson, S. B., Garcia-Molina, H., & Skeen, D. (1985). Consistency in a partitioned network: a survey. *ACM Computing Surveys*, *17*(3), 341–370. doi:10.1145/5505.5508

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., et al. (1987). Epidemic algorithms for replicated database maintenance. In PODC '87: *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, New York, NY, USA. ACM Press.

Demers, A. J., Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., & Welch, B. B. (1994). The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, 8-9.

Dollimore, J., Coulouris, G., and Kindberg, T. (2001). *Distributed Systems: Concepts and Design*. Pearson Education 2001, 3 edition.

Ericsson, A. B. (1998). Edge - introduction of high-speed data in gsm/gprs networks. http://www.ericsson.com/technology/whitepapers/.

Eshghi, K., Lillibridge, M., Wilcock, L., Belrose, G., & Hawkes, R. (2007). Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 22–22, Berkeley, CA, USA. USENIX Association.

Fekete, A., Gupta, D., Luchangco, V., Lynch, N. A., & Shvartsman, A. A. (1996). Eventually-serializable data services. In *Symposium on Principles of Distributed Computing*, pages 300–309.

Fidge, C. (1991). Logical time in distributed computing systems. *Computer*, *24*(8), 28–33. doi:10.1109/2.84874

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol http/1.1. Internet Request for Comment RFC 2616*. Internet Engineering Task Force.

Fitzpatrick, B. W., Pilato, C. M., & Collins-Sussman, B. (2004). *Version Control with Subversion*. O'Reilly.

Forman, G. H., & Zahorjan, J. (2001). The challenges of mobile computing. *Computer*, *27*(4), 38–47. doi:10.1109/2.274999

Fox, A., & Brewer, E. A. (1999). Harvest, yield, and scalable tolerant systems. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 174, Washington, DC, USA. IEEE Computer Society.

Golding, R. (1993). *Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases*. Technical Report UCSC-CRL-93-09, UC Santa Cruz.

3GPP. (n.d.). *3rd Generation Partnership Project*. Retrieved from http://www.3gpp.org/.

Guy, R. G., Reiher, P. L., Ratner, D., Gunter, M., Ma, W., & Popek, G. J. (1998). Mobile data access through optimistic peer-to-peer replication. In *ER Workshops* (pp. 254–265). Rumor.

Henson, V., & Garzik, J. (2002). *Bitkeeper for kernel developers*. http://infohost.nmt.edu/˜val/ols/bk.ps.gz.

Holliday, J., Steinke, R., Agrawal, D., & El Abbadi, A. (2003). Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, *15*(5), 1218–1238. doi:10.1109/TKDE.2003.1232274

IEEE. (1997). IEEE 802.11 Wireless Local Area Networks Working Group. http://grouper.ieee.org/groups/802/11/index.html.

Ignat, C.-L., & Norrie, M. C. (2006). Draw-Together: Graphical editor for collaborative drawing. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 269–278, Banff, Alberta, Canada.

Ignat, C.-L., Oster, G., Molli, P., Cart, M., Ferrié, J., Kermarrec, A.-M., et al. (2007). A comparison of optimistic approaches to collaborative editing of wiki pages. In *CollaborateCom*, pages 474–483. IEEE.

Jagadish, H. V., Mumick, I. S., & Rabinovich, M. (1997). Scalable versioning in distributed databases with commuting updates. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 520–531, Washington, DC, USA. IEEE Computer Society.

Jain, N., Dahlin, M., & Tewari, R. (2005). Taper: Tiered approach for eliminating redundancy in replica sychronization. In *USENIX Conference onf File and Storage Technologies (FAST05)*.

Jajodia, S., & Mutchler, D. (1990). Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, *15*(2), 230–280. doi:10.1145/78922.78926

Kawell, J. L., Beckhardt, S., Halvorsen, T., Ozzie, R., & Greif, I. (1988). Replicated document management in a group communication system. In *CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, page 395, New York, NY, USA. ACM Press.

Keleher, P. (1999). Decentralized replicated-object protocols. In Proc. Of the 18th Annual ACM Symp. on *Principles of Distributed Computing (PODC'99)*.

Kermarrec, A.-M., Rowstron, A., Shapiro, M., & Druschel, P. (2001). The IceCube approach to the reconciliation of divergent replicas. In *20th Symp. on Principles of Dist. Comp. (PODC)*, Newport RI (USA). ACM SIGACT-SIGOPS.

Kistler, J. J., & Satyanarayanan, M. (1991). Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. ACM SIGOPS.

Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., et al. (2000). Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558–565. doi:10.1145/359545.359563

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, *C-28*, 690–691. doi:10.1109/TC.1979.1675439

Lee, Y.-W., Leung, K.-S., & Satyanarayanan, M. (1999). Operation-based update propagation in a mobile file system. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX.

Lelewer, D. A., & Hirschberg, D. S. (1987). Data compression. *ACM Computing Surveys*, *19*(3), 261–296. doi:10.1145/45072.45074

Leuf, B., & Cunningham, W. (2001). *The wiki way: Quick collaboration on the web*. Addison-Wesley.

Levy, E., & Silberschatz, A. (1990). Distributed file systems: Concepts and examples. *ACM Computing Surveys*, *22*(4), 321–374. doi:10.1145/98163.98169

Lidl, K., Osborne, J., & Malcolm, J. (1994). Drinking from the firehose: Multicast usenet news. In *Proc. of the Winter 1994 USENIX Conference*, pages 33–45, San Francisco, CA.

MacDonald, J. (2000). *File system support for delta compression*. Masters thesis, University of California at Berkeley.

Malkhi, D., Novik, L., & Purcell, C. (2007). P2P replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, *41*(2), 68–74. doi:10.1145/1243418.1243427

Malkhi, D., & Terry, D. B. (2005). Concise version vectors in winfs. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of Lecture Notes in Computer Science, pages 339–353. Springer.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V.

Microsoft. (2000). *Windows 2000 Server: Distributed systems guide* (pp. 299–340). Microsoft Press.

Miltchev, S., Smith, J. M., Prevelakis, V., Keromytis, A., & Ioannidis, S. (2008). Decentralized access control in distributed file systems. *ACM Computing Surveys*, *40*(3), 1–30. doi:10.1145/1380584.1380588

Mockapetris, P. V., & Dunlap, K. J. (1995). Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, *25*(1), 112–122. doi:10.1145/205447.205459

Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S., & Smith, F. D. (1986). Andrew: a distributed personal computing environment. *Communications of the ACM*, *29*(3), 184–201. doi:10.1145/5666.5671

Muthitacharoen, A., Chen, B., & Mazieres, D. (2001). A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187.

Nowicki, B. (1989). *NFS: Network file system protocol specification. Internet Request for Comment RFC 1094*. Internet Engineering Task Force.

Paluska, J. M., Saff, D., Yeh, T., & Chen, K. (2003). Footloose: A case for physical eventual consistency and selective conflict resolution. In *5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 170–180, Monterey, CA, USA, October 9–10.

Pâris, J.-F., & Long, D. D. E. (1988). Efficient dynamic voting algorithms. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 268–275, Washington, DC, USA. IEEE Computer Society.

Pedone, F. (2001). Boosting system performance with optimistic distributed protocols. *Computer*, *34*(12), 80–86. doi:10.1109/2.970581

Pedone, F., Guerraoui, R., & Schiper, A. (2003). The Database State Machine Approach. *Distributed and Parallel Databases*, *14*(1), 71–98. doi:10.1023/A:1022887812188

Petersen, K., Spreitzer, M., Terry, D., & Theimer, M. (1996). Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland.

Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., & Demers, A. J. (1997). Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16)*, Saint Malo, France.

Peterson, Z. N. J., Burns, R., Ateniese, G., & Bono, S. (2007). Design and implementation of verifiable audit trails for a versioning file system. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 20–20, Berkeley, CA, USA. USENIX Association.

Quinlan, S., & Dorward, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA.

Ramsey, N., & Csirmaz, E. (2001). An algebraic approach to file synchronization. *SIGSOFT Softw. Eng. Notes*, *26*(5), 175–185. doi:10.1145/503271.503233

Ratner, D., Reiher, P., & Popek, G. (1999). Roam: A scalable replication system for mobile computing. In *DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 96,Washington, DC, USA. IEEE Computer Society.

Ratner, D. H. (1998). *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD Thesis 970044, University of California, 31.

Reiher, P., Popek, G., Cook, J., & Crocker, S. (1993). Truffles—a secure service for widespread file sharing. In *PSRG Workshop on Network and Distributed System Security*.

Rowstron, A., & Druschel, P. (2001). Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201.

Royer, E., & Toh, C.-K. (1999). A review of current routing protocols for ad hoc mobile wireless networks. *Personal Communications, IEEE*, *6*(2), 46–55. doi:10.1109/98.760423

Saito, Y., Bershad, B. N., & Levy, H. (1999). Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, New York, NY, USA. ACM Press.

Saito, Y., Karamanolis, C., Karlsson, M., & Mahalingam, M. (2002). Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Operating Systems Review*, 36(SI):15–30.

Saito, Y., & Shapiro, M. (2005). Optimistic replication. *ACM Computing Surveys*, *37*(1), 42–81. doi:10.1145/1057977.1057980

Santos, N., Veiga, L., & Ferreira, P. (2007). Vector-field consistency for ad-hoc gaming. In *ACM/IFIP/Usenix International Middleware Conference (Middleware 2007)*, Lecture Notes in Computer Science. Springer.

Santry, D., Feeley, M., Hutchinson, N., Veitch, A., Carton, R., & Ofir, J. (1999). Deciding when to forget in the elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA. ACM Press.

Schiper, N., Schmidt, R., & Pedone, F. (2006). Optimistic algorithms for partial database replication. In Alexander A. Shvartsman (Ed.), *OPODIS*, volume 4305 of Lecture Notes in Computer Science, pages 81–93. Springer.

Schwarz, R., & Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, *7*(3), 149–174. doi:10.1007/BF02277859

Shapiro, M., Bhargavan, K., & Krishna, N. (2004). A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Lecture Notes In Computer Science, pages 331–345, Grenoble, France.

Spring, N. T., & Wetherall, D. (2000). A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*.

Storer, M., Greenan, K., Miller, E., & Voruganti, K. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA. USENIX Association.

Strunk, J., Goodson, G., Scheinholtz, M., Soules, C., & Ganger, G. (2000). Self-securing storage: protecting data in compromised system. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 12–12, Berkeley, CA, USA. USENIX Association.

Sun, C., & Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, Seattle WA, USA.

Sutra, P., & Shapiro, M. (2008). Fault-tolerant partial replication in large-scale database systems. In *Europar*, pages 404–413, Las Palmas de Gran Canaria, Spain.

Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., & Hauser, C. H. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the* fifteenth ACM Symposium on Operating Systems Principles, pages 172–182. ACM Press.

Thomas, G., Thompson, G., Chung, C.-W., Barkmeyer, E., Carter, F., & Templeton, M. (1990). Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, *22*(3), 237–266. doi:10.1145/96602.96607

Torres-Rojas, F., & Ahamad, M. (1999). Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, *12*(4), 179–195. doi:10.1007/s004460050065

Trigdell, A., & Mackerras, P. (1998). *The rsync algorithm*. Technical report, Australian National University. http://rsync.samba.org.

Valot, C. (1993). Characterizing the accuracy of distributed timestamps. *SIGPLAN Not.*, *28*(12), 43–52. doi:10.1145/174267.174272

Walker, B., Popek, G., English, R., Kline, C., & Thiel, G. The locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70.

Weiser, M. (1991). The computer for the twenty-first century. *Scientific American*, *265*, 94–104. doi:10.1038/scientificamerican0991-94

Wessels, D., & Claffy, K. (1997). *Internet cache protocol. Internet Request for Comment RFC 2186*. Internet Engineering Task Force.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., & Alonso, G. (2000). Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C. IEEE Computer Society Technical Committee on Distributed Processing.

Wilson, P. (1991). *Computer Supported Cooperative Work: An Introduction*. Oxford: Intellect Books.

Wuu, G., & Bernstein, A. (1984). Efficient solutions to the replicated log and dictionary problems. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242, New York, NY, USA. ACM Press.

Yu, H., & Vahdat, A. (2000). Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of Operating Systems Design and Implementation*, pages 305–318.

Yu, H., & Vahdat, A. (2001). The costs and limits of availability for replicated services. In *Symposium on Operating Systems Principles*, pages 29–42.

Zhang, Y., Paxson, V., & Shenker, S. (2000). *The stationarity of internet path properties: Routing, loss, and throughput*. ACIRI Technical Report.

KEY TERMS AND DEFINITIONS

Optimistic Replication: Strategy for data replication in which replicas are allowed to diverge and consistency is achieved a posteriori.

Pessimistic Replication: Strategy for data replication in which any access to replicated data is only granted after the system guarantees that no inconsistency will result from such an access.

Conflict: Situation where two updates cannot be scheduled in any order that is safe, according to some application semantics.

Eventual Consistency: Paradigm that allows a replicated system to be temporarily inconsistent, while ensuring that eventually the system will agree on and converge to a state that is strongly consistent.

Commitment: System-wide agreement on a schedule of previously updates that are guaranteed to eventually be applied at a consistent order at any replica and to never roll back at any replica.

Partial Replication: Form of data replication that allows each replica to hold only a subset of the data items comprising the corresponding object.

Data Deduplication: Technique that avoids transferring or storing data that the receiver site already stores at some local object.

ENDNOTES

1 Except where noted, this chapter assumes *full replication*, i.e. each site that replicates a given object maintains a replica of the whole value of the object. Furthermore, we assume a full-trust model. Solutions relying on more realistic trust models for replicated systems can be found, for instance, in (Miltchev et al., 2008), (Reiher et al., 1993), (Kubiatowicz et al., 2000), (Rowstron & Druschel, 2001) or (Boulkenafed & Issarny, 2003).

2 In practice, the system may represent updates in various forms, as we discuss later in the chapter.

3 This is also called the prefix property (Petersen et al., 1997).

4 Since referential integrity of site identifiers is hardly solved by distributed protocols or centralized name servers.

5 In particular, the Database State Machine to partial replication approach (Pedone, Guerraoui & Schiper, 2003).