Principles of Eventual Consistency

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

now
the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

S. Burckhardt. *Principles of Eventual Consistency*. Foundations and Trends® in Programming Languages, vol. 1, no. 1-2, pp. 1–150, 2014.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-60198-859-1

Foundations and Trends® in Programming Languages
Volume 1, Issue 1-2, 2014

Editorial Board

Full text available at: http://dx.doi.org/10.1561/2500000011

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2014, Volume 1, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Foundations and Trends® in Programming Languages
Vol. 1, No. 1-2 (2014) 1–150

DOI: 10.1561/2500000011

now
the essence of knowledge

Principles of Eventual Consistency

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Contents

Abstract

In globally distributed systems, shared state is never perfect. When communication is neither fast nor reliable, we cannot achieve strong consistency, low latency, and availability at the same time. Unfortunately, abandoning strong consistency has wide ramifications. Eventual consistency, though attractive from a performance viewpoint, is challenging to understand and reason about, both for system architects and programmers. To provide robust abstractions, we need not just systems, but also principles: we need the ability to articulate what a consistency protocol is supposed to guarantee, and the ability to prove or refute such claims.

In this tutorial, we carefully examine both the what and the how of consistency in distributed systems. First, we deconstruct consistency into individual guarantees relating the data type, the conflict resolution, and the ordering, and then reassemble them into a hierarchy of consistency models that starts with linearizability and gradually descends into sequential, causal, eventual, and quiescent consistency. Second, we present a collection of consistency protocols that illustrate common techniques, and include templates for implementations of arbitrary replicated data types that are fully available under partitions. Third, we demonstrate that our formalizations serve their purpose of enabling proofs and refutations, by proving both positive results (the correctness of the protocols) and a negative result (a version of the CAP theorem for sequential consistency).

S. Burckhardt. *Principles of Eventual Consistency*. Foundations and Trends® in Programming Languages, vol. 1, no. 1-2, pp. 1–150, 2014.

DOI: 10.1561/2500000011.

1

Introduction

As our use of computers relies more and more on a complex web of clients, networks, and services, the challenges of programming a distributed system become relevant to an ever expanding number of programmers. Providing good latency and scalability while tolerating network and node failures is often very difficult to achieve, even for expert architects. To reduce the complexity, we need programming abstractions that help us to layer and deconstruct our solutions. Such abstractions can be integrated into a language or provided by some library, system API, or even the hardware.

A widely used abstraction to simplify distributed algorithms is *shared state*, a paradigm which has seen much success in the construction of parallel architectures and databases. Unfortunately, we know that in distributed systems, shared state cannot be perfect: in general, it is impossible to achieve both strong consistency and low latency. To state it a bit more provocatively:

> All implementations of mutable shared state in a geographically distributed system are either slow (require coordination when updating data) or weird (provide weak consistency only).

This unfortunate fact has far-reaching consequences in practice, as it forces programmers to make an unpleasant choice. Strong consistency means that reads and updates behave as if there were a single copy of the data only, even if it is internally replicated or cached. While strong consistency is easy to understand, it creates problems with availability and latency. And unfortunately, availability and latency are often crucial for business — for example, on websites offering goods for sale, any outage may cause an immediate, irrecoverable loss of sales [G. DeCandia et al., 2007]. Where business considerations trump programming complexity, consistency is relaxed and we settle for some form of

Eventual Consistency. The idea is simple: (1) replicate the data across participants, (2) on each participant, perform updates tentatively locally, and (3) propagate local updates to other participants asynchronously, when connections are available.

Although the idea is simple, its consequences are not. For example, one must consider how to deal with conflicting updates. Participants must handle conflicting updates consistently, so that they agree on the outcome and (eventually) converge. Exactly what that should mean, and how to understand and compare various guarantees, data types, and system implementations is what we study in this tutorial.

Although eventual consistency is compelling from a performance and availability perspective, it is difficult to understand the precise guarantees of such systems. This is unfortunate: if we cannot clearly articulate a specification, or if the specification is not strong enough to let us write proveably correct programs, eventual consistency cannot deliver on its promise: to serve as a robust abstraction for the programming of highly-available distributed applications.

The goal of this tutorial is to provide the reader with tools for reasoning about consistency models and the protocols that implement them. Our emphasis is on using basic mathematical techniques (sets, relations, and first order logic) to describe a wide variety of consistency guarantees, and to define protocols with a precision that enables us to prove both positive results (proving correctness of protocols) and negative results (proving impossibility results).

1.1 General Motivation

Geographical distribution has become inseparable from computing. Almost all computers in use today require a network connection to deliver their intended functionality. Programming a distributed system has thus become common place, and understanding both the challenges and the available solutions becomes relevant for a large number of programmers. The discipline of distributed computing is at the verge of a "relevance revolution" not unlike the one faced by concurrent and parallel computing a decade ago. Like the "multicore revolution", which forced concurrent and parallel programming into the mainstream, the "mobile+cloud revolution" means that distributed programming in general, and the programming of devices, web applications, and cloud services in particular, is well on its way to becoming an everyday necessity for developers. We can expect them to discover and re-discover the many challenges of such systems, such as *slow communication, scalability bottlenecks, and node and network failures*.

1.1.1 Challenges

The performance of a distributed system is often highly dependent on the *latency* of network connections. For technical and physical reasons (such as the speed of light), there exists a big disparity between the speed of local computation and of wide-area communication, usually by orders of magnitude. This disparity forces programmers to reduce communication to keep their programs performant and responsive.

Another important challenge is to achieve *scalability* of services. Scalability bottlenecks arise when too much load is placed on a resource. For example, using a single server node to handle all web requests does not scale. Thus, services need to be distributed across multiple nodes to scale. The limited resource can also be the network. In fact, it is quite typical that the network gets saturated by communication traffic before the nodes reach full utilization. Then, programmers need to reduce communication to scale the service further.

And of course, there are *failures*. Servers, clients, and network connections may all fail temporarily or permanently. Failures can be a

consequence of imperfect hardware, software, or human operation. The more components that there are in a system, the more likely it will fail from time to time, thus failures are unavoidable in large-scale systems.

Often, it makes sense to consider failures not as some rare event, but as a predictable part of normal operation. For example, a connection between a mobile client and a server may fail because the user is driving through a tunnel or boarding an airplane. Also, a user of a web application may close the browser without warning, which (from a server perspective) can be considered a "failure" of the client.

At best, failures remain completely hidden from the user, or are experienced as a minor performance loss and sluggish responses only. But often, they render the application unusable, sometimes without indication about what went wrong and when we may expect normal operation to resume. At worst, failures can cause permanent data corruption and loss.

1.1.2 Role of Programming Languages

What role do programming languages have to play in this story? A great benefit of a well-purposed programming language is that it can provide convenient, robust, and efficient abstractions. For example, the abstraction provided by a garbage-collected heap is convenient, since it frees the programmer from the burden of explicit memory management. It is also robust, since it cannot be broken inadvertently if used incorrectly. Last but not least (and only after much research on the topic), garbage collection is efficient enough to be practical for many application requirements. Although conceptually simple, garbage collection illustrates what we may expect from a successful combination of programming languages and systems research: a separation of concerns. The client programmer gets to work on a simpler abstracted machine, while the runtime system is engineered by experts to efficiently simulate the abstract machine on a real machine.

But what abstractions will indeed prove to be convenient, robust, and efficient in the context of distributed systems? Ideally, we would like to completely hide the distributed nature of the system (slow connections, failures, scalability limits) from the programmer. If we could

efficiently simulate a non-distributed system on a distributed system, the programmer would never even need to know that the system is distributed. Unfortunately, this dream is impossible to achieve in general. This becomes readily apparent when we consider the problem of *consistency of shared state*. In a non-distributed system, access to shared data is fast and atomic. However, the same is not true for a distributed system.

1.1.3 Distributed Shared Data

Ideally, simulating shared data in a distributed system should look just like in a non-distributed system - meaning that it should appear as if there is only a single copy of the data being read and written.

The Problem. There is no doubt that strong consistency (also known as single-copy consistency, or linearizability) is the best consistency model from the perspective of application programmers. Unfortunately, it comes at a cost: maintaining the illusion of a single copy requires communication whenever we read or update data. This communication requirement is problematic when connections are slow or unavailable. Therefore, any system that guarantees strong consistency is susceptible to the following problems:

- **Availability.** If the network should become partitioned, i.e. if it is no longer possible for all nodes to communicate, then some clients may become unusable because they can no longer update or read the data.

- **Performance.** If each update requires a round-trip to some central authority, or to some quorum of servers or peers, and if communication is slow (for example, because of geographical distance between the client and the server, or between the replicas in a service), then the performance and responsiveness of the client application suffers.

These limitations of strong consistency are well known, and complicate the design of many distributed applications, such as cloud services.

The CAP theorem, originally conjectured by Brewer [2000] and later proved by Gilbert and Lynch [2002], is a particularly popular formulation of this fundamental problem (as discussed in the IEEE Computer retrospective edition 2012). It states that strong **C**onsistency and **A**vailability cannot be simultaneously achieved on a **P**artitioned network, while it is possible to achieve any combination of two of the above properties.

Seat Reservation Example. We can illustrate this idea informally using an example where two users wish to make an airplane reservation when there is only one seat left. Consider the case where the two users reside in different network partitions, and are thus incapable of communicating in any way (even indirectly through some server). It is intuitively clear that in such a situation, any system is forced to delay at least one user's request, or perhaps both of them (thus sacrificing availability), or risk reserving the same seat twice (thus sacrificing consistency). Achieving both availability and consistency is only possible if the network always allows communication (thus sacrificing partition tolerance).

This simple seat reservation example is a reasonable illustration of the hard limits on what can be achieved. However, it may also create an overly pessimistic and narrow view of what it means to work with shared state in a distributed system. Airlines routinely overbook seats, and reservations can be undone (at some cost). The real world is not always strongly consistent, for many more reasons than just technological limitations.

1.2 Applications

Practitioners and researchers have proposed the use of eventual consistency to build more reliable or more responsive systems in many different areas.

- **Cloud Storage and Georeplication.** Eventual consistency can help us to build highly-available services for cloud storage, and to keep data that is replicated across data centers in sync. Examples include research prototypes [Li et al., 2012, Lloyd et al.,

2011, 2014, Sovran et al., 2011] and many commercially used storage systems such as Voldemort, Firebase, Amazon Dynamo [G. DeCandia et al., 2007], Riak [Klophaus, 2010], and Cassandra [Lakshman and Malik, 2009].

- **Mobile Clients.** Eventual consistency helps us to write applications that provide meaningful functionality while disconnected from the network, and remain highly responsive even if connections to the server are slow [Terry et al., 1995, Burckhardt et al., 2012b, 2014b].

- **Epidemic or Gossip Protocols**. Eventual consistency can help us to build low-overhead robust monitoring systems for cloud services, or for loosely connected large peer-to-peer networks [Van Renesse et al., 2003, Jelasity et al., 2005, Princehouse et al., 2014].

- **Collaborative editing.** When multiple people simultaneously edit the same document, they face consistency challenges. A common solution is to use operational transformations (OT) [Imine et al., 2006, Sun and Ellis, 1998, Nichols et al., 1995].

- **Revision Control.** Forking and merging of branches in revision control system is another example where we can apply general principles regarding concurrent updates, visibility, and conflict resolution [Burckhardt and Leijen, 2011, Burckhardt et al., 2012a].

The examples above span a rather wide range of systems. The participating nodes may have little computational power and storage space (such as mobile phones) or plenty of computation power (such as servers in data centers) and lots of storage (such as storage back-ends in data centers). Similarly, the network connections may be slow, unreliable, low-bandwidth and expensive (e.g. cellular connections) or fast and high-bandwidth (e.g. intra-datacenter networks), or something in between (e.g. inter-datacenter networks). These differences are very important when considering how best to make the trade-off between reliability and availability. However, at an abstract level, all of these sys-

tems share the same principles of eventual consistency: shared data is updated at different replicas, updates are transmitted asynchronously, and conflicts are resolved consistently.

1.3 Warmup

To keep things concrete, we start with a pair of examples. We study two different implementations of a very simple shared data type, a register. The first one stores a single copy on some reliable server, and requires communication on each read or write operation. The second one propagates updates lazily, and both read and write operations complete immediately without requiring communication.

For illustration purposes, we keep the shared data very simple: just a value that can be read and written by multiple processes. This data type is called a register in the distributed systems literature. One can imagine a register to be used to control some configuration setting, for example.

1.3.1 Single-Copy Protocol

The first implementation of the register stores a single copy of the register on some central server — it does not use any replication. When clients wish to read or write the register, they must contact the server to perform the operation on their behalf. This general design is very common; for example, web applications typically rely on a single database backend that performs operations on behalf of clients running in web browsers.

We show the *protocol definition* in Fig. 1.1. A protocol definition specifies the name of the protocol, the messages, and the *roles*. The SingleCopyRegister protocol defines four messages and two roles, Server and Client.

Roles represent the various participants of the protocol, and are typically (but not necessarily) geographically separated. Roles react to operation calls by some user or client program, and they communicate with each other by sending and receiving messages. Technically, each role is a state machine which defines a current state and *atomic*

```
1  protocol SingleCopyRegister {
2
3    message ReadReq(cid: nat) : reliable
4    message ReadAck(cid: nat, val: Value) : reliable
5    message WriteReq(cid: nat, val: Value) : reliable
6    message WriteAck(cid: nat) : reliable
7
8    role Server {
9      var current: Value;
10     receive(req: ReadReq) {
11       send ReadAck(req.cid, current);
12     }
13     receive(req: WriteReq) {
14       current := req.val;
15       send WriteAck(req.cid);
16     }
17   }
18
19   role Client(cid: nat) {
20     operation read() {
21       send ReadReq(cid);
22      // does not return to client program yet
23     }
24     operation write(val: Value) {
25       send WriteReq(cid,val);
26      // does not return to client program yet
27     }
28     receive ReadAck(cid) {
29       return val; // return to client program
30     }
31     receive WriteAck(cid) {
32       return ok; // return to client program
33     }
34   }
35 }
```

Figure 1.1: A single-copy implementation of a register. Read an write operations contact the server and wait for the response.

transitions that are executed in reaction to operation calls by client programs, to incoming messages, or to some periodic scheduling. In our notation, roles look a bit like objects: the role state looks like fields of an object, and each atomic transition looks like a method of the object.

A role definition starts with the name of the role, followed by an argument list that clarifies the number of instances, and how they are distinguished. Here, there is a single server role and an infinite number of clients, each identified by a client identifier cid which is a nonegative integer (type nat).

Messages. There are four message format specifications (lines 3 – 6). Each one describes a message type and the contents of the message (names and types), and specifies the expected level of reliability. For example, the declaration **message** WriteReq(c: Client, val:boolean) : reliable means that each WriteReq message carries a client identifier c (the client writing the register), and a boolean value val (the value being written), and that this message is always delivered to all recipients, and never forged nor duplicated, but possibly reordered with other messages.

Server. In the Server role (lines 8 – 17), the state of the server consists of a single variable current which is the current value of the register (line 9). It is specified to be initially false. The only server actions are to receive a read or a write request. When receiving a message corresponding to a read request (line 10) or a write request (line 13), the corresponding operation (read or write) is performed, and the result value (in the case of read) or an acknowledgment message (in the case of write) is sent back using a **send** request.

Client. The Client role (lines 19 – 34) contains definitions for read and write operations, but has no variables (*i.e.* it is *stateless*). Supposedly, the operations are called by the local user or client program; the latter may call any sequence of read and write operations, but it may not call an operation until the previous one has returned.

When the read operation is called, the corresponding atomic transition sends a WriteReq message, but it does *not* complete the operation — there is no implicit return at the end of a transition (the opera-

tion cannot return because it does not know the value of the register yet). Only when the response arrives from the server, the corresponding transition contains an explicit **return** statement that completes the read operation and returns the result to the client program. Thus, the read-operation is non-atomic, i.e. executes not as a single transition, but as two transitions. The write operation is non-atomic as well; it blocks until an acknowledgment from the server has been received.

Message Destination. Note that the send instruction does not explicitly specify the destination — instead, it is the receive instruction that specifies what messages to receive. Receive operations specify a *pattern* that defines what messages can be received.[1] For example, the receive actions on lines 28 and 31 match an incoming message only if the c field of the request matches this, which is the client id — therefore, only the c field acts as a destination identifier and ensures the response message is received only by the client that sent the original request to the server.

Atomic Actions. Our semantics compiles roles like state machines with atomic actions. Intuitively, this means that only one block of code is executing at a time, thus there is no fine-grained concurrency and we need no locks. Of course, there is still ample opportunity for subtle errors caused by the coarse-grained concurrency, *i.e.* by unexpected orderings of the atomic actions.

Reliability. Crashes by one client cannot impact other clients. However, the protocol is not robust against server crashes: a crashed server makes progress impossible for all clients. This assumption of a single reliable server is of course the cornerstone of the single-copy protocol design. It is, however, not a limitation of the epidemic protocol defined in the next section.

```
1  protocol EpidemicRegister {
2
3    struct Timestamp(number: nat; pid: nat);
4    function lessthan(Timestamp(n1,pid1), Timestamp(n2,pid2)) {
5      return (n1 < n2) ∨ (n1 == n2 ∧ pid1 < pid2);
6    }
7
8    message Latest(val: Value, t: Timestamp) : dontforge, eventualindirect
9
10   role Peer(pid: { 0 .. N }) {
11
12     var current: Value := undef;
13     var written: Timestamp := Timestamp(0,pid);
14
15     operation read() {
16       return current;
17     }
18     operation write(val: Value) {
19       current := val;
20       written := Timestamp(written.number + 1,pid);
21       return ok;
22     }
23
24     periodically {
25        send Latest(current, written);
26     }
27
28     receive Latest(val,ts) {
29       if (written.lessthan(ts)) {
30         current := val;
31         written := ts;
32       }
33     }
34   }
35 }
```

Figure 1.2: An implementation of the register where all operations return immediately, without waiting for messages.

1.3.2 Epidemic Protocol

The single-copy implementation is easy to understand. However, the read and write operations are likely to be quite slow in practice because they require a round-trip to the server. The epidemic register (Fig. 1.2) eliminates this problem by removing the server communication from the operations: each role stores a local copy of the register, and propagates updates asynchronously. No central server is needed: all roles are equal (we call them peers). We call this a *symmetric* protocol, as opposed to the asymmetric client-server protocol discussed in the previous section.

Timestamps. When propagating updates, we use timestamps to ensure that later updates overwrite earlier ones and not the other way around. Each node stores not just the currently known latest value of the register (current), but also a timestamp (written) that indicates the time of the write operation that originally wrote that value. When receiving a timestamped update, we ignore it if its timestamp is older than the timestamp of the current value.

Logical clocks. Rather than a physical clock, we use *logical clocks* to create timestamps, which are a well-known, clever technique for ordering events in a distributed system [Lamport, 1978]. Logical timestamps are pairs of numbers, which are totally ordered by lexicographic order[2] as defined on lines 3–5. On each write operation (lines 18–22) the node creates a new timestamp, which is larger than the current one (and thus also larger than all timestamps previously received in update messages).

Update Propagation. Every once in a while, each role performs the code on lines 24–26 which broadcasts the currently stored value and its timestamp in a Latest message. This ensures that all roles become eventually aware of all updates, and are thus eventually consistent.

[1] These patterns are similar to patterns in languages like OCaml, but must be static, i.e. the pattern may not depend on the current state of the role, but must use only constants.

[2] Lexicographic order means that tuples are compared based on the first component, and then the second component if the first one is the same, and so on. It is a generalization of alphabetic order if we consider words to be tuples of letters, thus the name.

Weaker Delivery Guarantees. The delivery guarantees required by this protocol (on line 8) are dontforge (meaning no messages should be invented) and eventualindirect (meaning that there must be some delivery path, possibly indirect via other replicas). These are weaker conditions than the reliable guarantee used by the single-copy protocol (which required that all messages be delivered to all receivers exactly once). Here, the system is allowed to duplicate and even lose messages, as long as there is always eventually some (possibly indirect) delivery path from each sender to each receiver.

This type of propagation is sometimes called *epidemic*, since nodes can indirectly "infect" other nodes with information. An epidemic protocol keeps functioning even if some connections are down, as long as the topology is "eventually strongly connected". Another name for this type of protocol is *state-based*, because each message contains information that is identical to the local state.

Consistency and Correctness

The interesting questions are: is the epidemic protocol correct? What does correct even mean? What is the observable difference between the two protocols, from a client perspective?

Given our discussion of eventual consistency earlier, we may reasonably expect an answer along the lines of "the epidemic protocol is eventually consistent, while the single-copy protocol is strongly consistent". However, the story is a bit more interesting than that.

- The single-copy register is *linearizable*, which is the strongest form of consistency.
- The epidemic register is *sequentially consistent*, which is a slightly weaker, yet still surprisingly strong consistency guarantee. We prove this in §10.2.2.

At first glance, this appears to contradict the CAP theorem since the epidemic register is available under partitions (all operations complete immediately), thus strong consistency should not be possible? It turns out that the original CAP is about linearizability, not sequential

consistency; and under sequential consistency, CAP only applies to reasonably expressive data types, not including a simple register. We prove a properly qualified version of the CAP theorem in §9.1.2.

Since the single-copy register is linearizable, and the epidemic register is sequentially consistent, they are observationally equivalent to any client that does not have a side channel for communication (for more about this, see §5.3.1).

1.4 Overview

The goal of this tutorial is to provide the reader with tools for reasoning about consistency of protocols. Our emphasis is on using basic mathematical techniques (sets, relations, and first order logic) to describe a wide variety of consistency guarantees, and to define protocols with a level of precision that enables us to prove both positive results (correctness of protocols) and negative results (refute implementability).

We start with basic technical foundations in chapter 2, including a review of important concepts related to partial and total orders. We also introduce *event graphs*, which are mathematical objects representing information about events in executions, and which are the technical backbone of all our definitions.

In chapters 3–5, we lay out the specification methodology, and assemble consistency guarantees spanning data type semantics, ordering guarantees, and convergence guarantees:

- In chapter 3 we introduce our approach to specifying consistency guarantees, which is based on histories and abstract executions.
- In chapter 4, we first specify the semantics of sequential data types, and then generalize to *replicated data types* that specify the semantics in a replicated setting, in particular how to resolve conflicts. The key insight is to think of the current state not as a value, but as a graph of prior operations.
- In chapter 5, we define basic eventual consistency, collect various consistency guarantees, and present a hierarchy of the most common consistency models.

In chapter 6, we walk through a selection of protocol implementations and optimizations, to gain a better understanding of the nature of the trade-off between the consistency model and the speed/availability of operations. We show implementations for simple data types, and protocol templates that can be used to implement any replicated data type.

In chapters 7 and 8, we establish formal models for executions in asynchronous distributed systems (including crashes and transport failures), and for protocol definitions (accommodating arbitrary asynchronous protocols). These models are needed as a preparation for the next two chapters, which conclude the technical development:

- In chapter 9, we prove a version of the CAP theorem that shows that for all but the simplest data types, sequential consistency cannot be implemented in a way such that all operations are available under partitions.
- In chapter 10, we revisit the implementations presented earlier, and prove that they provide the claimed consistency guarantees.

References

IEEE Computer CAP retrospective edition. *Computer*, 45(2), 2012.

Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, February 2012.

Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2), 1991.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 204–219. Springer, 2012.

Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

Sarita Adve and Mark Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993. ISSN 1045-9219. .

Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, 1985. .

Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering (ICSE)*, pages 562–570. IEEE, 1976.

Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley, 1998.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, not CAP: Towards highly available transactions. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2013.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *Very Large Data Bases (VLDB)*, 2014.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*, 2011.

Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Principles of Programming Languages (POPL)*, 2013.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *International Conference on Management of Data (SIGMOD)*, pages 1–10, 1995.

Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *International Conference on Management of Data (SIGMOD)*, pages 923–928. ACM, 2013.

Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Technical Report 8083, INRIA, 2012a.

Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *Conference on Distributed Computing (DISC)*, 2012b.

Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Programming Language Design and Implementation (PLDI)*, pages 68–78, 2008.

Eric A. Brewer. Towards robust distributed systems (abstract). In *Principles of Distributed Computing (PODC)*, 2000.

Sebastian Burckhardt. *Memory model sensitive analysis of concurrent data types*. PhD thesis, University of Pennsylvania, 2007.

Sebastian Burckhardt and Daan Leijen. Semantics of Concurrent Revisions. In *European Symposium on Programming (ESOP), LNCS*, volume 6602, pages 116–135, 2011.

Sebastian Burckhardt, Rajeev Alur, and Milo M.K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Impl. (PLDI)*, pages 12–21, 2007.

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *Programming Language Design and Implementation (PLDI)*, pages 330–340, 2010.

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012a.

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012b.

Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft, 2013.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Principles of Programming Languages (POPL)*, 2014a.

Sebastian Burckhardt, Daan Leijen, and Manuel Fahndrich. Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43, Microsoft, 2014b.

Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper. *Replication*, volume 5959 of *LNCS*. Springer, 2010.

William W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-767187-3.

Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer Aided Verification (CAV)*, LNCS 4144, pages 475–488. Springer, 2006.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2003.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

David L. Dill. The murphi verification system. In *Computer Aided Verification (CAV)*, pages 390–393. Springer-Verlag, 1996.

Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

Alan D. Fekete and Krithi Ramamritham. Consistency models for replicated data. In *Replication*, volume 5959 of *LNCS*, pages 1–17. Springer, 2010. .

Colin J. Fidge. Timestamps in message passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of The ACM*, 1982.

G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Utah, 2005.

Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33: 51–59, June 2002.

T. Lockman Greenough. *Representation and Enumeration of Interval Orders*. PhD thesis, Dartmouth College, 1976.

Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *Conference on Distributed Computing (DISC)*, LNCS 2180, pages 300–314. Springer, 2001.

Tim Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. In *Conference on Distributed Computing (DISC)*, LNCS 2508, pages 265–279. Springer, 2002.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems (OPODIS)*, 2005.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3): 463–492, 1990.

Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. *TCS*, 351(2), 2006.

Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3): 219–252, August 2005.

Rusty Klophaus. Riak core: Building distributed applications without shared state. In *Commercial Users of Functional Programming (CUFP)*. ACM SIGPLAN, 2010.

Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, June 2012.

Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

Leslie Lamport. Introduction to TLA. Technical Report SRC Technical Note 1994-001, Digital Equipment Corporation, 1994.

Cheng Li, Daniel Porto, Allen Clement, Rodrigo Rodrigues, Nuno Preguiça, and Johannes Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *Operating Systems Design and Implementation (OSDI)*, 2012.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Systems Design and Implementation (NSDI)*, pages 313–328. USENIX, 2013.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual consistency. *Commun. ACM*, 57, 2014.

Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Principles of Distributed Computing (PODC)*, pages 137–151. ACM, 1987.

Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, 2005.

Edward Marczewski. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16:386–389, 1930.

Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1989.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *User interface and software technology (UIST)*, 1995.

Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *Operating Systems Review*, 31:288–301, 1997.

Lonnie Princehouse, Rakesh Chenchu, Zhefu Jiang, Ken Birman, Nate Foster, and Robert Soule. Mica: A compositional architecture for gossip protocols. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

James B. Rothnie and Nathan Goodman. A survey of research and development in distributed database management. In *Very Large Data Bases (VLDB)*, pages 48–62, 1977.

Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37:42–81, 2005. .

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Programming Language Design and Implementation (PLDI)*, pages 175–186. ACM, 2011.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of EATCS*, (104), 2011a.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011b.

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2011c.

Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2): 282–312, 1988. ISSN 0164-0925. .

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *CSCW*, 1998.

Douglas B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool, May 2008. ISBN 9781598292022.

Douglas B. Terry. Replicated data consistency explained through baseball. (MSR-TR-2011-137), October 2011.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems (PDIS)*, 1994.

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symposium on Operating Systems Principles (SOSP)*, 1995.

Robert H. Thomas and Bolt Beranek. A Majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4:180–209, 1979.

V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 129–136, 2006.

Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

Jason Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, University of Utah, 2005.